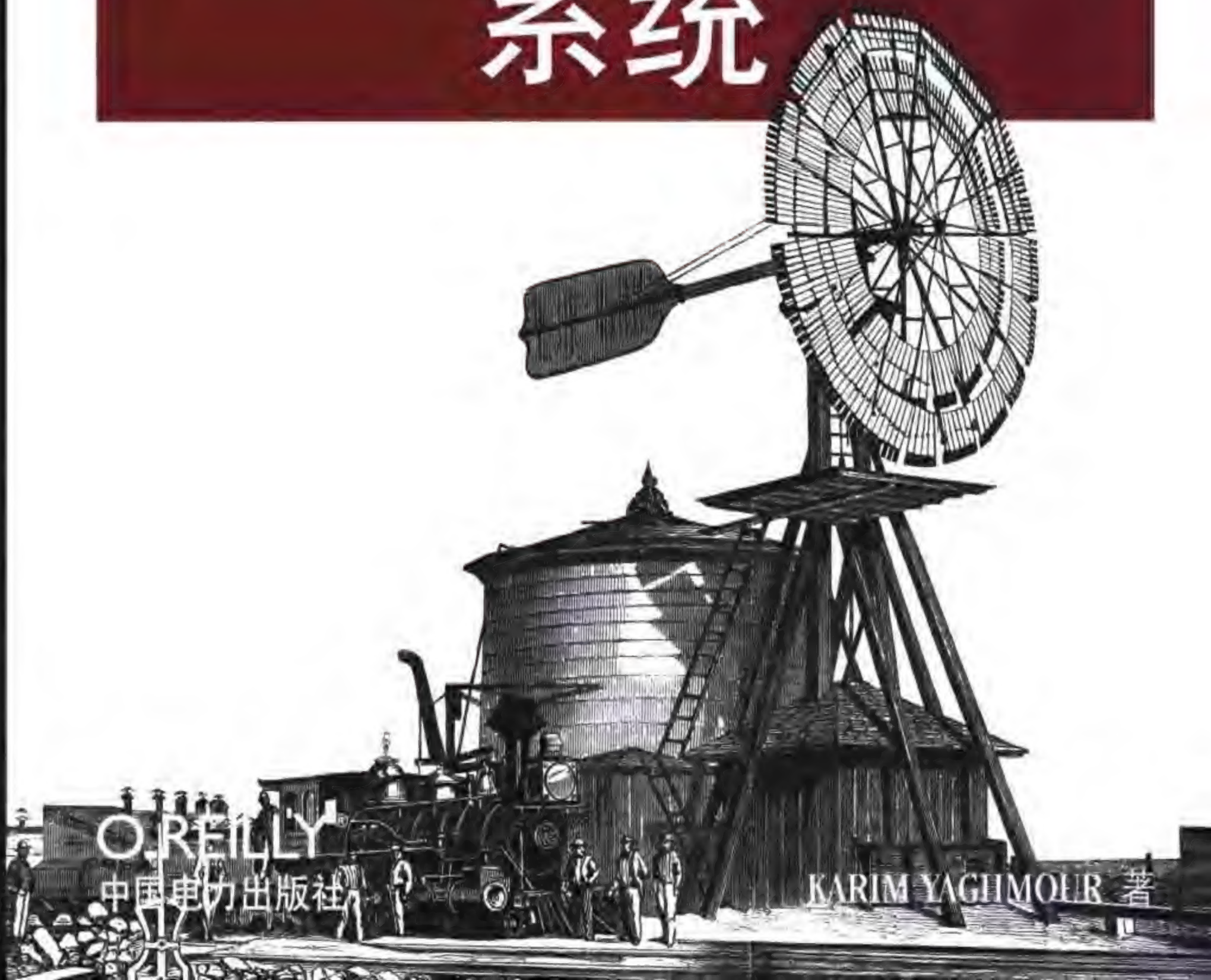


BUILDING EMBEDDED LINUX SYSTEMS

构建嵌入式 LINUX 系统



O'REILLY

中国电力出版社

KARIM YAGHMOUR 著

构建嵌入式Linux系统



《构建嵌入式Linux系统》是业界第一本深入探讨基于Linux内核的嵌入式系统开发的权威指南。这本不可或缺的书总结了下面这些过程中的秘诀，这些过程甚至以前从未形成过正式文档：

- 建立自己的GNU开发工具链
- 为特定目标板选择、配置、建立以及安装内核
- 建立完整的目标板根文件系统
- 设置、管理以及使用固态存储设备
- 为目标板安装并配置引导加载程序
- 使用多种工具和技术调试嵌入式系统

本书详细解释了多种不同的目标板架构和硬件配置，包括彻底分析支持嵌入式硬件的Linux。所有的解释都针对的是开源和自由软件包。通过演示如何从源码建立操作系统组件，以及如何查找更多文档和帮助。本书极大地简化了完全控制嵌入式操作系统的任务，不管是基于技术还是经济上的原因。

作者Karim Yaghmour是一位著名的设计者和演讲者，他负责Linux Trace Toolkit。本书从讨论Linux作为嵌入式操作系统的优缺点开始，其中包括了许可证问题。然后讨论了构建嵌入式Linux系统的基础知识，随后的讨论涵盖了嵌入式Linux系统中广泛使用的40余种开源和自由软件包的配置、设置以及使用。uClibc、BusyBox、U-Boot、OpenSSH、tftp、ifip、strace以及gdb都在讨论范围之内。

“如果你正使用或开发嵌入式Linux系统，或者打算未来使用或开发嵌入式Linux系统，你应该购买本书。本书写得很好，信息量大，并且不回避难題，比如如何建立工具链，或者如何远程调试应用程序这样的难题。我确实认为本书将会成为嵌入式Linux世界中的《Linux设备驱动程序》……强烈推荐。”

——Erik Andersen, uClibc、BusyBox和TinyLogin的主要开发者和维护者

ISBN 7-5083-2754-3



9 787508 327549 >

O'REILLY®

www.oreilly.com.cn

O'Reilly Media, Inc. 授权中国电力出版社出版

ISBN 7-5083-2754-3

定价：48.00元

构建嵌入式 LINUX 系统

Karim Yaghmour 著
O'Reilly Taiwan 公司 译
韩存兵 龚波 改编

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc. 授权中国电力出版社出版

中国电力出版社

图书在版编目 (CIP) 数据

构建嵌入式 Linux 系统 / (美) 雅默 (Yaghmour, K.) 著; 韩存兵, 龚波改编.
— 北京: 中国电力出版社, 2004

书名原文: Building Embedded Linux Systems

ISBN 7-5083-2754-3

I. 构 ... II. ①雅 ... ②韩 ... ③龚 ... III. Linux 操作系统 IV. TP316.89

中国版本图书馆 CIP 数据核字 (2004) 第 112453 号

北京市版权局著作权合同登记

图字: 01-2005-0824 号

©2003 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Electric Power Press, 2004. Authorized translation of the English edition, 2003 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版 2003。

简体中文版由中国电力出版社出版 2004。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有, 未得书面许可, 本书的任何部分和全部不得以任何形式重制。

书 名 / 构建嵌入式 Linux 系统

书 号 / ISBN 7-5083-2754-3

责任编辑 / 陈维宁

封面设计 / Emma Colby, 张健

出版发行 / 中国电力出版社 (www.infopower.com.cn)

地 址 / 北京三里河路 6 号 (邮政编码 100044)

经 销 / 全国新华书店

印 刷 / 北京市地矿印刷厂

开 本 / 787 毫米 × 1092 毫米 16 开本 26.5 印张 426 千字

版 次 / 2004 年 12 月第一版 2004 年 12 月第一次印刷

印 数 / 0001-4000 册

定 价 / 48.00 元 (册)

O'Reilly Media, Inc. 介绍

为了满足读者对网络 and 软件技术知识的迫切需求,世界著名计算机图书出版机构 O'Reilly Media, Inc. 授权中国电力出版社, 翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司, 同时是联机出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》(被纽约公共图书馆评为二十世纪最重要的 50 本书之一) 到 GNN (最早的 Internet 门户和商业网站), 再到 WebSite (第一个桌面 PC 的 Web 服务器软件), O'Reilly Media, Inc. 一直处于 Internet 发展的最前沿。

许多书店的反馈表明, O'Reilly Media, Inc. 是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比, O'Reilly Media, Inc. 具有深厚的计算机专业背景, 这使得 O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员, 或者是顶尖级的技术专家。O'Reilly Media, Inc. 还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家, 而现在编写著作, O'Reilly Media, Inc. 依靠他们及时地推出图书。因为 O'Reilly Media, Inc. 紧密地与计算机业界联系着, 所以 O'Reilly Media, Inc. 知道市场上真正需要什么图书。

目录

前言	1
第一章 概述	11
定义	11
嵌入式 Linux 系统的实际应用	14
以多组件系统为例	39
设计与实现方法	45
第二章 基本概念	50
主机类型	50
主机 / 目标板开发设置的类型	52
主机 / 目标板调试设置的类型	55
嵌入式 Linux 系统的一般架构	56
系统启动过程	58
引导配置的类型	59
系统存储器的设计	62
第三章 所支持的硬件	64
处理器架构	64
总线与接口	71

I/O	81
存储设备	91
通用网络	99
工业级网络	107
系统监控	111

第四章 开发工具 113

实际项目工作空间的使用	113
GNU 跨平台开发工具链	115
C 链接库的替代品	140
Java	148
Perl	151
Python	153
Ada	154
其他程序语言	155
集成开发环境	156
终端仿真程序	157

第五章 内核方面的考虑 163

选择内核	163
内核配置	166
编译内核	171
安装内核	173
实地测试	175

第六章 根文件系统的内容 178

根文件系统的基本结构	178
链接库	182
内核模块	187
内核映像	187

设备文件	188
主要的系统应用程序	190
定制应用程序	198
系统初始化	198

第七章 存储设备管理 205

MTD 支持的设备	205
磁盘设备	229
是否启用交换功能	231

第八章 根文件系统的设置 232

选择文件系统	232
使用经 NFS 安装的根文件系统将文件系统映像写入 flash 设备	236
CRAMFS	237
JFFS2	240
NFTL 上的磁盘文件系统	241
RAM disk 上的磁盘文件系统	242
安装 TMPFS 上的目录	244
在线更新	245

第九章 设置 bootloader 254

各式各样的 bootloader	255
网络引导的服务器设置	261
在磁盘和 CompactFlash 设备上使用 LILO	266
在 DiskOnChip 设备上使用 GRUB	269
U-Boot	273

第十章 设置网络服务 293

Internet Super-Server	293
使用 SNMP 进行远程管理	297

通过 Telnet 进行网络登录	300
使用 SSH 进行安全通信	303
通过 HTTP 提供 Web 内容	309
通过 DHCP 进行动态配置	313
第十一章 调试工具.....	315
用 gdb 进行应用程序调试	315
跟踪	322
性能分析	330
内存调试	339
关于硬件工具	342
附录一 工作单	345
附录二 资源	361
附录三 重要的版权声明	365
源代码索引	377



前言

我在1997年任职一家硬件厂商的时候，首次提议在嵌入式系统使用 Linux，我的建议在当时不仅令人感到惊讶，也受到某些质疑。现在，在嵌入式系统使用 Linux 不再是可笑的事情了。的确，有不少大公司及政府机关在其嵌入式软件的需求上逐渐开始依赖 Linux。

Linux 在一些嵌入式应用上的成功应用引起了广泛的兴趣和热烈的回应。这导致了“嵌入式 Linux”相关文章、网站、公司及文档的泛滥。除了一闪即逝的宣布，杂志里的文章，数以百计的项目和产品，都宣告了将 Linux 用于嵌入式系统是轻而易举的事情，然而正在探索有用指南的专业开发者们，却仍在为建立嵌入式 Linux 系统的基本方法和技术寻觅答案。

目前大部分的文档都只着重于如何使用各类套件即跨平台开发工具，以及目标板二进制码 (target binary，译注 1)。有的文档则只局限在如何在目标板 (target board) 上运行 Linux。

与现有的文档不同，本书并不会假定你手边有哪些工具，或是限定项目的范围。本书只会要你通过 Internet 下载必要的套件，浏览特定的在线文档，以及通过项目的邮件论坛与其他开发者一起分享经验。你还需要开发主机以及目标板硬件的说明文件，不会要求你向任何厂商购买任何产品或服务。

译注 1：目标板 (target) 是嵌入式系统的代名词，通常用于软件开发期间，用来区分嵌入式系统与开发主机。目标板二进制码 (target binary) 指的是目标板 (target board) 上以二进制 (0 与 1) 形式存在内存中供 CPU 执行的机器码 (machine code) 或目标码 (object code)。

除了提供最大的自由度并让你掌控自己的设计之外,这么做还最接近在嵌入式系统中使用Linux的先驱者们所采用的方式。这些先驱者对Linux所做的事,本质上包括将它拆开以符合应用的需要,并根据其目的对它进行定制。因此,Linux突破嵌入式世界的方式与许多软件厂商将其产品扩展至新应用领域的做法不同。作为嵌入式系统开发者,你可能会发现,与厂商销售的产品相比,Linux比较容易应用到设计中。

本书的做法就是提供所有的细节,并讨论在嵌入式系统中使用Linux时将会遇到的各种困境,让你能够把Linux应用在设计中。虽然本书的内容不可能涵盖所有的嵌入式设计,但是书中提供的资源却可以让你轻易获得必要的信息,使你能够在嵌入式系统中使用Linux,并对它进行定制。

撰写本书的目的在于拉近在设计中使用开放源码和自由软件的嵌入式系统的开发者跟建立和维护这些开放源码和自由软件套件的开发者的距离。尽管有许多主流嵌入式系统开发者(其中不乏高水平的程序设计者)会依赖第三方提供他们需要的嵌入式Linux,但无疑他们也有机会对所依赖的开放源码和自由软件(free software)计划做出贡献。最后,这种动力将可确保Linux持续成为嵌入式系统首选的操作系统。

本书对象

本书锁定的第一种读者,是那些想要在未来或目前的计划中使用Linux的有经验的嵌入式系统设计者。本书会假定这一类读者已经熟悉用于开发嵌入式系统的技术和术语,如交叉编译,进行BDM或JTAG调试,以及如何处理不完善或不完全的硬件。如果是这一类读者,可能会想要跳过前面一些章节中与嵌入式系统开发背景知识有关的部分。然而,还是需要阅读一些前面的章节(尤其是第二章),因为这一章包含了在嵌入式系统中使用Linux内核的特别提示。

本书锁定的第二种读者,是那些想要熟悉嵌入式Linux系统开发工具和技术但没有经验的嵌入式系统开发者。本书并非嵌入式系统的入门书,不过若你想探究本书所讨论到的一些课题,可以参考一些入门性质的教科书。本书附录二列出了一些有用的书籍以及信息来源。

如果你是对Linux已经很熟悉的高级用户或系统管理员,那么本书应该可以帮助你进行高度定制的Linux安装。例如,如果你发现发行套件安装了过多的包,而你想要从头开始建立自己定制的发行套件,本书有许多部分应该可以派上用场,特别是第六章的内容。

最后,本书应该可以帮助那些想要了解如何建立与操作Linux系统的程序设计者或Linux的热衷者。尽管本书并未说明如何建立一般用途的发行套件,不过其所提到的许多技术,在一定程度上可以媲美用来进行定制嵌入式Linux安装的一般用途发行套件。

本书范畴及背景知识

要在嵌入式系统中展现 Linux 的最佳性能，需要以下背景知识，这些知识在许多书中都交代得很清楚：

嵌入式系统

一般而言，你必须熟悉嵌入式系统的开发、程序设计和调试，这包括软件、硬件两方面。

Unix 系统管理

你必须能够胜任各种系统管理工作，如硬件的配置、系统的设定、维护以及利用 shell 命令脚本让管理工作自动化。

Linux 设备驱动程序

你必须知道如何对各种 Linux 设备驱动程序进行开发及调试。

Linux 内核的运行原理

你必须尽可能了解内核的运行原理。

GNU 软件开发工具

你必须具备有效利用 GNU 工具的能力。这包括了解许多常被认为是晦涩难懂的选项和工具程序。

本书会假定你至少熟悉论述题目中的基本概念。另一方面，阅读本书你不需要知道一些内容，例如 Linux 设备驱动程序是如何建立的，或与嵌入式系统开发有关的每一件事。翻阅本书的时候，若看到符合你的嵌入式系统的 Linux 用法，可能你会觉得需要取得这个 Linux 用法的进一步信息。除了阅读本书时你自己参考的其他书籍，还可以看一看附录二所列的书单，或许会从中找到这些背景知识的进一步信息。

尽管本书只讨论如何在嵌入式系统中使用 Linux，但是对想要在嵌入式系统中使用 BSD 的开发者来说多少会有一些帮助。不过本书所作的许多说明都必须依据 BSD 与 Linux 间的差异重新诠释。

本书架构

本书由三大部分构成。第一部分由第一到三章组成。这三章的内容涵盖建立任何嵌入式 Linux 系统必备的基本知识。尽管这三章并未提到任何程序，不过它们都是建立嵌入式 Linux 系统时不可或缺的知识。

第二部分由第四到九章组成。这几章的内容都非常重要，其中安排了用来建立任何嵌入式 Linux 系统的必要步骤。不管你的系统的目的或功能是什么，都必须阅读这几章的内容。

第三部分由第十和十一章组成。虽然这两章所包含的内容都很重要，不过并不是建立嵌入式 Linux 系统必需的。

第一章“概述”将会深入嵌入式 Linux 系统的世界。本章首先会介绍基本定义，接着会探讨嵌入式 Linux 系统的实际问题，包括从嵌入式系统的观点来探讨开放源码和自由软件版权问题。然后会介绍本书其他部分将会用到的范例系统，以及本书将会用到的实现方法。

第二章“基本概念”用来描述建立任何嵌入式 Linux 系统时必须具备的基本概念。

第三章“所支持的硬件”完整地介绍了 Linux 所支持的嵌入式硬件，以及告诉读者何处可以找到实现这些硬件的驱动程序和子系统。本章还会探讨处理器的架构、总线和接口、I/O、存储设备、一般用途的网络功能、工业等级的网络功能、以及系统监控。

第四章“开发工具”将会说明如何安装与使用各种用来建立嵌入式 Linux 系统的开发工具。其中最值得注意的是，如何从头开始建立与安装 GNU 工具链组件。此外还会分节讨论 Java、Perl 和 Python，以及探讨可用来与嵌入式目标板交互的各种终端仿真程序。

第五章“内核方面的考虑”将会探讨在嵌入式系统中 Linux 内核的选择、配置、交叉编译、安装以及使用。

第六章“根文件系统的内容”将会说明如何使用本书前面提到的组件来建立根文件系统，包括 C 链接库的安装，以及建立适当的 */dev* 条目。比较值得注意的是，本章的说明涵盖了 BusyBox、TinyLogin、Embutils 和 System V *init* 的安装和使用。

第七章“存储设备管理”将会讲解如何操作以及为嵌入式 Linux 系统配置存储设备。本章将重点放在固态存储设备上，如固有的 flash 和 DiskOnChip 设备，以及 MTD 子系统。

第八章“根文件系统的设置”将会说明如何为嵌入式系统的存储设备设置在第六章建立的根文件系统。这包括建立 JFFS2 和 CRAMFS 文件系统的映像，以及在 NFTL 上使用磁盘形式的文件系统。

第九章“设置 bootloader”将会针对每个嵌入式 Linux 平台探讨各种可用的 bootloader（引导加载程序）。并且会进一步探讨如何在 DiskOnChip 设备上使用 GRUB 以及 U-Boot。此外还包含用 BOOTP/DHCP、TFTP 和 NFS 来进行网络引导。

第十章“设置网络服务”会将重点放在各种网络服务（如SNMP、SSH和HTTP）的软件包的配置、安装和使用。

第十一章“调试工具”的内容涵盖了嵌入式Linux系统开发软件时可能遇到的主要调试问题。这包含了如何在跨平台（cross-platform）开发环境中使用gdb、进行跟踪、性能分析，以及内存调试。

附录一“工作单”提出了一份可搭配本书一起使用的工作单，可以为嵌入式Linux系统提供完整的计划书。

附录二“资源”的内容包含了建立嵌入式Linux系统时可供利用的资源。

附录三“重要的版权声明”的内容包含了Linus Torvalds与其他内核开发者讨论内核版权、以及非GPL版权的二进制内核模块相关问题时的重要信件。

尽管第七到九章看起来像是各自独立的章节，不过它们的内容却高度地关联。举例来说，要了解第七章探讨的目标板存储设备的配置，得先了解第八章探讨的目标板文件系统，反之亦然。此外，存储设备的设置还必须先了解第九章探讨的bootloader的设置与操作，反之亦然。因此，我建议各位先一口气读完第七到九章，再执行这几章中提到的任何指令。这样当你设定目标板的时候，才有办法按照这几章所描述的程序进行操作。

本书用到的硬件

正如我们在第三章中所见，Linux支持各式各样的硬件。由于篇幅有限，本书只能选用几种嵌入式系统作为各种程序的测试平台。表P-1完整地列出了本书将会用到的系统。

这些系统中有些是主流市场上的商品，如iPAQ或Dreamcast。我刻意将它们列入，主要是让有兴趣的读者可以轻易找到用来学习建立嵌入式Linux系统的材料。例如，Sega出品的Dreamcast游戏机，可以在eBay上以低于50美元的价格买到。尽管Dreamcast使用特殊格式的CD-ROM来引导，不过它却是学习Linux跨平台开发的最便宜方法之一。当然，也可以使用旧的x86 PC来作实验，不过这样就比较没意思了。

表 P-1: 本书所用到的目标板系统

架构	系统类型	处理器的 时钟速度	RAM的大小	存储空间的 大小及类型
PPC	TQ components TQM860L	80 MHz	16 MB	8 MB flash
SuperH	Sega Dreamcast	200 MHz	16 MB	CD-ROM（见正文）
ARM	Compaq iPAQ 3600	206 MHz	32 MB	16 MB flash

表 P-1: 本书所用到的目标板系统 (续)

架构	系统类型	处理器的 时钟速度	RAM 的大小	存储空间的大小及类型
x86	Kontron Teknor VIPer 806	100 MHz	40 MB	32 MB CompactFlash
x86	COTS ^a Pentium	100 MHz	8 MB	32 MB DiskOnChip

a. 现成的商品。

除了可以在嵌入式系统上运行，Linux 还可以在各种工作站上运行。本书将会用到的主机如表 P-2 所示。尽管 Apple PowerBook 是本书主要的开发主机，但是在建立基于 x86 的嵌入式目标板时，还是会用到 x86 主机，因为有些软件组件无法在非 x86 的主机上为 x86 目标板进行交叉编译。举例来说，GRUB 和 LILO 就必须建立在 x86 主机上。然而，可以告诉各位的是，这整本书是在执行 Yellow Dog Linux 发行套件的 PowerBook 主机上写成的。另一个具有象征意义的是，Linux 的版图已经从计算机世界延伸到硬件这个非常不完整的世界了。

表 P-2: 本书用到的主机系统

架构	系统类型	处理器的时钟速度	RAM 的大小	存储空间的大小
PPC	Apple PowerBook	400 MHz	128 MB	> GB 硬盘
x86	Pentium II	350 MHz	128 MB	> GB 硬盘

为了说明可以使用 Linux 的各种目标板架构，我在各章列举的范例会刻意使用不同的目标板硬件。表 P-3 列出了各章用到的目标板架构。尽管各章的说明基于不同的架构，不过各章提到的命令却可同时应用到其他架构上。例如，若某章所举的例子中需要用到 *arm-linux-gcc* 命令（用于 ARM 的 *gcc* 编译器），则只要将它替换成 *powerpc-linux-gcc* 命令，就可以将这个例子应用到 PPC 架构上。表 P-3 中，若某章列出的架构不止一个，则探讨的主要架构会列在第一个。例如，第五章所举的命令范例会将重点放在 ARM 架构上，与 PPC 架构有关的命令只会提到几个。

虽然表 P-3 列出了各章范例中用到的目标板架构，但该表并未指出其使用的开发主机，这是因为使用哪种主机进行开发并无差别。所以除非特别声明，否则主机的架构总是不同于目标板的架构。举例来说，虽然第四章所提到的是以 PPC 主机为 x86 目标板建立工具，不过只要稍做或不做修改就可以在 SPARC 或 S/390 上面执行相同指令。请注意，本书前几章大部分的内容与架构无关，所以不需要提供任何架构的专用命令。

表 P-3: 各章范例用到的主要目标板架构

章号	目标板架构
第一章	无
第二章	无
第三章	无
第四章	x86
第五章	ARM、PPC
第六章	PPC
第七章	x86、PPC
第八章	ARM
第九章	PPC、x86
第十章	ARM
第十一章	PPC

软件的版本

当然,嵌入式Linux系统依靠的中心软件就是Linux内核。本书将焦点放在2.4版的Linux内核上,尤其是编号2.4.18的发行版本。以2.4版为依据所做的变更大概只会影响到本书前面几章所提供的一些信息。也就是说,新发行版本所支持的硬件大概会比第三章所列的还多。不过就2.4版来说,本书提到的基本任务不太可能会变更。不过当内核的发展越过2.4版进入新的里程碑(亦即2.6版)时,本书提到的某些步骤就有可能需要更新了。

此外,本书探讨了40种以上开放源码及自由软件包的配置、安装和使用。每种软件包的维护各自独立,而且开发步调也各不相同。因为这些软件包会随着时间的推移而发生变化,所以有可能本书提到的软件包的版本等到读者阅读本书的时候已经过时了。为了降低软件更新对内容造成的影响,本书会尽可能让内容与版本无关。由于与软件的变动无关,所以全书的结构和各章的结构不太可能需要改动。再者,本书提到的许多软件包已经问世一段时间了,所以它们不太可能会有什么重大的变动。例如,本书所使用的GNU开发工具链,其中各组件的安装、设置和使用命令,相对来说,已经好多年没什么变动了,所以未来也不太可能变动。这同样适用本书所提到的大多数软件包。

本书网站

本书提到的软件包有许多都还在持续开发中，这可能会导致必须修正本书的部分内容。于是我自己建立了一个网站为读者提供更新信息以及与本书有关的链接：

<http://www.embeddedtux.org/>

举例来说，可以到本书网站下载附录一提到的工作单，有PDF或OpenOffice两种格式。

字体约定

等宽字体 (Constant Width)

用来表示程序代码的内容或命令的输出以及程序代码中的关键词。

等宽粗体 (**Constant Width Bold**)

用来表示用户所键入的文字。

斜体 (*Italic*)

用来表示命令行选项、网址，以及文件、目录、程序和命令的名称。

建议与问题

O'Reilly 是世界性的计算机信息出版公司。我们永远乐意听到读者对出版物的意见，包括如何让本书可以更好的建议、指正本书的错误、或是读者建议本书改版时应该再加进来的其他主题。以下是本公司的联系资料：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

100080 北京市海淀区知春路 49 号希格玛公寓 B 座 809 室
奥莱理软件（北京）有限公司

与本书有关的在线信息（包括勘误、范例程序、相关链接）：

<http://www.oreilly.com/catalog/belinuxsys/index.html>

如果要对本书进行评论或询问技术问题，请发邮件到：

bookquestions@oreilly.com

info@mail.oreilly.com.cn

最后，您可以在 WWW 上找到我们：

<http://www.oreilly.com>

<http://www.oreilly.com.cn>

致谢

“E quindi uscimmo a riveder le stelle”（注 1）是意大利诗人但丁在《神曲》第一部《地狱篇》的结语。这句话可能会让读者误以为我撰写本书时没有一点喜悦之情。但丁的这句话明确地表达了我完成你手上这本书当时的感觉。特别是我必须承认在嵌入式系统中使用 Linux 的信息多如牛毛、要加以整理谈何容易，更别说将它们一股脑儿全都塞进一本书来告诉读者建立嵌入式 Linux 系统的实际方法。还好，我背后有许多有能力的人愿意帮助我。

首先要感谢我的编辑 Andy Oram。就像帮助但丁游历灵界的弗吉尔，Andy 带领我渡过撰写本书的每个阶段。除了别的帮助之外，他还耐心地纠正我不规范的成语，以确保我的文章不会辞不达意，并且铁面无私地指出哪些章节的内容不够深入。你现在之所以会读到比较好的文章，这都得益于 Andy 的指教。同样地，我要感谢 Ellen Siever，本书一开始是我跟她一起合作的。但是我们的合作关系却在本书完成之前提早结束了，有许多点子最后之所以能够成功放入本书，都要归功于她建设性的建议。

我真地非常幸运，本书能够遇到这么一个杰出的审阅者团队，我非常感谢他们付出大量的时间和精力阅读、改正以及指出本书的种种问题。他们是 Erik Andersen、Wolfgang Denk、Bill Gatliff、Russell King、Paul Kinzelman、Alessandro Rubini、David Schleef 和 David Woodhouse。其中，我特别要感谢 Alessandro 对完美的执着追求。本书接下来的内容，如果还能找到任何错误，毫无疑问地都是我的问题。

撰写关于如何在嵌入式系统中使用 Linux 的书，必须用到许多不同的硬件。这些嵌入式硬件的价钱通常很贵，我要感谢那些提供设备给我使用的公司和个人。我特别要感谢 Kontron 公司（<http://www.kontron.de/>）的 Sthane Martin 提供了一块 Teknor VIPer 806 实验版给我；DENX Software Engineering 公司（<http://www.denx.de/>）的 Wolfgang Denk

注 1：“离开此地，只见闪耀灿烂的满天繁星。”

提供了一块 TQ components TQM860L PPC 实验版给我；以及 Zee2 公司 (<http://www.zee2.com/>) 的 Steve Papacharalambous 和 Stuart Hughes 提供了一个 uCdimmm 系统给我。

身为开放源码和自由软件社群的用户与贡献者，我屡次受益于社群中其他成员共享出来的知识与成果，这鼓舞并促成了我撰写本书的意愿。基于这个原因，我要感谢许多人。首先，我要感谢 Michel Dagenais 教授的信任与指导，让我有机会毫无拘束地去探索这个未经探勘的领域 (uncharted terrain)。Linux Trace Toolkit 的开发 (这是我硕士论文的一部分)，让我与开放源码和自由软件社群的关系越来越密切。身为其中的一员，我遇到了许多洞察力卓越的人，并得到了许多帮助，我非常感激他们。他们是 Jacques Gélinas、Richard Stallman、Jim Norton、Steve Papacharalambous、Stuart Hughes、Paolo Mantegazza、Pierre Cloutier、David Schleeff、Wolfgang Denk、Philippe Gerum、Loic Dachary、Daniel Phillips 和 Alessandro Rubini。

最后，我一定要说，我欠 Sonia 一份人情，是她异于常人的耐心，让我得以无后顾之忧地将全部的时间投入本书的测试与撰写。由于她的支持与关心让本书得以顺利完成。“La main invisible qui a écrit les espaces entre les lignes est la sienne et je lui en suis profondément reconnaissant.” (注 2)。

注 2: “这本书的字里行间里充满了你的倩影，感谢你为我所做的一切。”

第一章

概述



自从1991年首次公开发表以来，Linux的应用范围就愈来愈广泛。Linux的使用最初仅局限在Internet上的开发者和热衷者所构成的松散团队，而今却成为了一个稳固的类Unix操作系统，可以应用在工作站、服务器以及群集上。Linux的成长与普及，加快了自由软件基金会（Free Software Foundation, FSF）所发起的工作，并且激起了日后所谓的开源运动。Linux一直都能够吸引媒体及商业的兴趣，这有助于确立Linux在操作系统界的地位。

说也奇怪，Linux原本与嵌入式系统毫无关系，而今却摇身一变，成为优选的操作系统。现在，在我们的生活中，从移动电话到医疗设备，包括航空导航系统、自动柜员机、MP3播放器、打印机、汽车以及许多我们还不知道的其他设备，到处都能看到嵌入式系统。环顾四周，每当看到内含微处理器的设备，多半会发现它又是另一个嵌入式系统。

在阅读本书的时候，你或许会感到奇怪，怎么会有人想要在嵌入式系统中使用Linux。是因为它的灵活性、它的健壮性、它的价格，有社群发展它，或是有众多厂商支持它。事实上，有许多理由会让人们想要用Linux来构建嵌入式系统，而且有许多方法可以完成这项工作。本章将会借着讨论定义、现实生活中的问题、嵌入式Linux系统的一般结构、范例及方法为读者提供本书其余内容的背景知识。

定义

Linux、嵌入式Linux和实时Linux等术语的用法常会有所不同。这些术语有时用得十分精确，有时却很含糊。接下来，让我们看看，在不同情况下这些术语分别代表什么含义。

什么是 Linux

Linux 常用来指 Linux 内核、Linux 系统或 Linux 发行套件等术语。大体而言，当我们要对非技术人员讲述这些术语时，使用 Linux 会比较有利，但如果要提供技术性的说明，就要多费口舌了。例如，如果我说：“Linux 具备连接 TCP/IP 网络的能力。”我指的到底是 Linux 内核里的 TCP/IP 栈，还是包含在 Linux 发行套件中的 TCP/IP 工具程序，还是二者？这种含糊不清的描述，实际上已成为“GNU/Linux”拥护者抨击的依据，他们指出：Linux 只是内核，但整个系统大部分都建立在 GNU 软件之上（译注 1）。

严格来说，Linux 指的是 Linus Torvalds 维护的（及通过主要和镜像网站发布的）内核。Linux 的程序代码库只有内核，绝对不包含工具程序。内核就是整个操作系统的核心。内核或许不是系统执行的第一个软件，因为引导加载程序可能会在它之前先执行。一旦内核执行之后，直到系统关机，都不会被置换出主存储器或被移除。实际上，内核会控制所有硬件，并对系统执行的其他软件提供较高级别的硬件抽象，例如进程、socket 以及文件。

因为内核不断在更新，所以必须使用一种编号方案来区分特定的发行版本。这个编号机制用点号相隔的三个数字来区分各个发行版本。头两个数字用来指定版本编号（译注 2），第三个数字用来指定发行编号。举例来说，Linux 2.4.20 的版本编号是 2.4，而发行编号是 20。奇数的次版本号，如 2.5，用来指定开发版内核。偶数的次版本号，如 2.4，用来指定稳定版内核。通常应该为嵌入式系统使用最新发行的稳定版内核。

以上的说法太过简略。事实上，Internet 上随处可以找到修改过的 Linux 内核，它们附带额外的版本信息。以 2.4.18-rmk3-hh24 为例，这个修改过的内核是 Familiar 计划发布的（<http://familiar.handhelds.org/>）。这是基于 2.4.18 版的 Linux 内核，不过它包含了额外的、由 Familiar 开发团队控制的版本编号-rmk3-hh24。这个额外的版本编号及内核本身，将会在第五章作更进一步的探讨。

Linux 还可以用来指运行 Linux 内核的硬件系统。如果朋友说，他的开发团队在公司最近的产品中使用了 Linux，他的意思或许不是单指内核。一个 Linux 系统必定包含内核，不过也必然会包含一些在 Linux 内核上运行的其他软件组件。这些软件组件通常是 GNU 软件的一部分，如 C 链接库及一些可执行的工具程序。也可能包括 X window 系统、或附加的实时功能，如 RTAI（<http://www.aero.polimi.it/~rtai/>）。

译注 1：参见《Free as in Freedom》一书第五章“Small Puddle of Freedom”（<http://www.oreilly.com/openbook/freedom/ch05.html>）。

译注 2：第一个数字是主版本号，第二个数字是次版本号。

正如稍后所见，Linux 系统是可以定制的，或是使用既有的发行套件。你朋友的开发团队或许会定制自己的系统。相对而言，当有个用户说，她在计算机上执行 Linux，她的意思多半是指，她安装了某种 Linux 发行套件，如 Red Hat 或是 Debian。其他用户的 Linux 系统与你朋友的 Linux 系统相比不会少什么东西，但内核除外，他们系统各自的用途可能不同，采用的软件套件也可能不同，运行的应用程序也可能不同。

最后，Linux 还可以用来指 Linux 发行套件。Red Hat、Mandrake、SuSE、Debian、Slackware、Caldera、MontaVista、Embedix、BlueCat、PeeWeeLinux 等等都是 Linux 发行套件。尽管它们在用途、规模和价钱上可能会有所差异，不过它们却拥有共同的目标：将文件和安装程序打包，让用户能够为特定的目的，将内核及各种应用程序安装在特定的硬件上。大多数人熟悉的是 CD-ROM 形式的 Linux 发行套件，这跟你从网站下载、解开以及根据文档进行安装没什么两样。主流的面向用户的发行套件与这种发行套件的差异在于前者会自动进行安装。

从下一章开始一直到本书结束，我将会避免使用“Linux”这个词。取而代之的是，我将会直接指明所探讨的对象。也就是说，我不会说“Linux 内核”，而会直接说“内核”；我不会说“Linux 系统”，而会直接说“系统”；我不会说“Linux 发行套件”，而会直接说“发行套件”。在以上这些情况之下，“Linux”这个字都会被隐而不宣，以避免造成任何混淆。不过我仍然会适时地使用“Linux”这个字眼，来称呼环绕在内核周围大量的软件和硬件资源。

什么是嵌入式 Linux

我们再次以 Linux 所代表的三个涵意开始：一个内核、一个系统、一个发行套件。我们应该立即把“内核”从这份清单中移除，因为 Linus 并未发行过嵌入式版本的内核。但这并不是指无法嵌入该内核，只是意味着建立嵌入式系统不需要特别的内核。通常，可以使用官方发行的内核来建立自己的系统。有时，可能会想要使用由第三方发行的、修改过的内核；这类内核为了特殊的硬件配置或支持特定的应用，经过特别的裁剪。例如，各种嵌入式发行套件提供的内核，通常会包含一些主内核源码树中找不到的优化程序代码，以及用来支持某些调试工具（如内核调试器）的补丁。然而，嵌入式系统中使用的内核与工作站或服务器上使用的内核主要的不同还是在于用来建立内核的配置。第五章将会提到建立内核的过程。

一个嵌入式 Linux 系统只是代表它是一个基于 Linux 内核的嵌入式系统，这并不意味着该内核使用了任何特定的链接库或用户工具。

一个嵌入式 Linux 发行套件可能包括：用来开发嵌入式 Linux 系统的平台、各种为了在嵌入式系统中使用而裁剪过的应用软件，或是这二者。

开发平台包含了各种用来协助开发嵌入式系统的开发工具。这可能包括特殊的源码浏览器、交叉编译器、调试器、项目管理软件、引导映像生成器，等等。这些发行套件会被安装在开发主机上。

经过裁剪的嵌入式发行套件提供了一组可以在嵌入式系统中使用的应用程序。这可能包括可以在目标板上使用的特殊链接库、可执行文件以及配置文件。也可能会为目标系统提供一种方法来简化根文件系统的产生。

因为本书探讨的就是嵌入式 Linux 系统，所以不必为提到的每个名称都冠以“嵌入式 Linux”这个字眼。因此，我将会把用来开发嵌入式 Linux 系统的主机称为“主机系统”，或简称为“主机”。嵌入式系统本身将会被我称为“目标板系统”或简称为“目标板”。发行套件提供的“开发平台”将会被我称为“开发发行套件”（注 1）。发行套件提供的经过裁剪的软件包则会被我称为“目标板发行套件”。

什么是实时 Linux

最初，实时 Linux 是指 1996 年在 Victor Yodaiken 管理下，由 Michael Barabanov 发表的 RTLinux 计划。这个计划的目标是在 Linux 环境下提供确定的响应时间。

不过，现在有更多的计划在 Linux 元下提供这种形式或其他形式的实时响应功能。RTAI、Kurt 和 Linux/RK 等计划都是在 Linux 环境下提供实时能力。有些计划是在 Linux 下插入第二个内核来增强实时响应能力。有些则借助补丁来增强 Linux 内核的实时响应能力。

Linux 加上“实时”这个形容词之后，代表的意义就有些不同了。它主要是用来描述系统或其中的组件具有固定的响应时间，但是严格来说，你可能会发现它提供的未必是我们需要的实时。我将会探讨“实时”问题，并会在“时限”这一节更进一步地定义“实时”这个形容词。

嵌入式 Linux 系统的实际应用

Linux 会被用来建立哪些类型的嵌入式系统？人们为什么要选用 Linux？在嵌入式系统中使用 Linux 会引发什么问题？人们如何使用它？当仔细考虑是否要在嵌入式系统中使用

注 1：将它称为“主机发行套件”（host distributions）也很好，不过正如稍后所见，有些开发者会选择直接开发目标，因此使用“开发发行套件”比较好。

Linux 时，心中便会浮现以上这些问题，或更多的问题。要建立这样的系统，首先必须为这些根本的问题找到满意的答案。这些答案不仅是泛泛的说明，它们将能够有助于你说服管理层，帮助你销售产品，最重要的是，让你能够评估能否达到最初的期望。

嵌入式 Linux 系统的类型

我们可以使用传统的市场划分方式划分嵌入式系统，比如航空与航天、汽车系统、消费性电子产品、电信等等类型。但是这种划分方式将无法为所指的系统提供额外的信息，因为嵌入式 Linux 系统可能会被构建成市场划分无法归类的样子。倒不如采用能够提供实际信息的评判标准，比如规模、时限、网络能力以及与用户交互的程度，对嵌入式系统进行分类。

规模

嵌入式系统的规模由一些不同的因素来决定。首先，是它实际的尺寸大小。有些系统的尺寸可能相当大，如那些用来构建集群的产品。有些却相当小，如 IBM 制造的 Linux 手表。最重要的是，嵌入式系统的规模与系统中各个电子元件的属性有关，如 CPU 的运算速度、RAM 的容量、永久性存储器的容量。

就“规模”来说，嵌入式系统大致可以分成三类：小型、中型、大型。小型系统的特性是：低运算能力的 CPU，并且可以使用至少 2 MB 的 ROM 和 4MB 的 RAM。这并不表示说，Linux 无法在更小的内存空间中运行，只是你得多费些功夫就是了。如果计划在更小的内存空间中运行 Linux，那些将 Linux 放在一张软盘上的发行套件，可以作为起点。倘若你有嵌入式系统的背景，可能会发现，在这种小型的系统中，不使用 Linux 而采用其他操作系统反倒可以做更多的事。切记，在你部署 Linux 的时候，一定要考虑 CPU 的速度。

中型系统的特性是：中等运算能力的 CPU，并且可以使用大约 32MB 的 ROM 和 64MB 的 RAM。大多数以 Linux 构建的消费性产品皆属此类。这包括 PDA、MP3 播放器、娱乐系统以及网络设备。这些产品有些可能会包含如下形式的辅助存储器：固态硬盘（译注 3）、快闪存储卡，或者甚至是传统的硬盘。这些设备拥有足够的马力和存储容量来处理各种小型任务，或者为需要大量资源的单一任务提供服务。

大型系统的特性是：使用运算能力强大的 CPU，或是使用多个 CPU，并且可以使用大量的 RAM 及永久性存储器。通常，这些系统会用在需要大量计算来完成特定任务的环境

译注 3：以 RAM 芯片制成的硬盘。

中。大型的电信交换机以及飞行仿真器皆属此类。这类系统的特点是，没有经费和资源上的限制。它们的设计重点在于产品的功能，至于价钱、规模和复杂度则是次要的问题。

你可能觉得奇怪，为何不让Linux在32位以下的处理器上运行。因为这个限制将传统应用在嵌入式系统上的许多处理器排除在外。事实上，以传统的嵌入式系统标准来看，所有运行Linux的系统都将会被归类为大型系统。当你拿它与只能使用4K存储器的8051相比时，这是绝对正确的。但请记住，目前的趋势是：处理器的速度越来越快、RAM的容量越来越大且便宜、系统尽可能被整合在一起，而且价格越来越低。对处理能力的要求提高了，也连带增加了对系统的最低需求，于是这类可以执行Linux的系统，很快就变成了标准。不过，在某些情况下，8位微控制器仍然是最好的选择。

16 位 Linux?

严格地讲，以上所说，Linux无法在32位以下的处理器上运行，并非完全准确。已经有人将Linux移植到一些奇怪的处理器上。例如，位于<http://elks.sourceforge.net/>的Embeddable Linux Kernel Subset（嵌入式Linux内核子集，ELKS）计划，其目标就是在16位处理器（如Intel 8086和286）上运行Linux。然而，内核上及用户空间上已开发完成的大量应用程序，仍以32位为主。因此，如果选择在低于32位的处理器上运行Linux，将会孤立无援。

时限

对嵌入式系统来说，有两种时限类型：严格和宽松。严格时限的要求是，系统必须在预定的时间之内作出反应，否则会发生灾难事件。我们以工厂中工人使用的大型材料切割机具为例，为了安全起见，会在刀具周围安装光传感器，以便侦测工人手上所戴具有特殊颜色的手套。当系统察觉工人的手有危险时，必须立即让刀具停止运转。没有时间等待CPU置换文件或撤销任务。这种系统具有严格的时间要求：也就是所谓的硬实时系统。

音频流系统对时间也有严格的要求，因为只要有任何的延迟通常都会令用户感到不舒服。然而，这种系统一般被归类为软实时系统，因为该系统一时的失误，并不会像硬实时系统那样，会造成无法弥补的灾难。换言之，尽管可以容忍这种系统发生一时的失误，不过这种系统还是应该设计为具备严格的时间要求。

宽松时限的要求有各种的形式，不过它们一般会被应用在需要及时（或适时）做出响应但不必立即完成的系统上。如果自动柜员机得花10多秒的时间才能完成交易，这一般不

算是什么问题。PDA启动应用程序需要数秒的时间也是同样的情况。需要额外的处理时间让系统看来似乎很慢，但并不会影响最后的结果。

网络能力

网络能力用来定义一个系统是否可以连上网络。当今，我们可以预期的是，通过网络能够访问一切事物，甚至是电冰箱。然而，这会在我们所建立的系统上产生特殊的需求。促使人们选用Linux作为嵌入式操作系统的一个因素，就是用来证明它的网络能力。下跌的价格及网络联机组件的标准化，都加快了这个趋势。大部分的Linux设备都具备某种形式的网络能力。例如，只需在PCMCIA背夹或扩充槽中插上适配器就可以在为Compaq iPAQ建立的Linux发行套件中加入无线网卡。

与用户的交互

与用户交互的程度，不同的系统之间会有非常大的差异。有些系统，如PDA的主要功能就是与用户交互。有些系统，如工业的过程控制，用于交互的部分可能只有LED（发光二极管按钮。有些系统甚至没有用户界面。例如，飞机自动驾驶系统中的某些部件会自己控制机翼，并不会直接与驾驶员交互。

实例

想要了解嵌入式Linux系统能做哪些事，最好的办法就是参考实际的例子。问题是，如果我们试着在Internet上寻找嵌入式系统公开的细节，几乎都只能找到消费性电子产品。Linux在航空与航天、工业控制、电信、汽车系统等方面的应用有公开细节的实例非常少，仿佛Linux尚未在这些应用中使用。相对于消费性电子产品，这些设备的制造商并不觉得公开他们的设计有什么好处。他们只知道，这么做可能会让那些想要进入Linux领域赶上他们的竞争者知道关键信息。反过来说，消费性电子产品的制造商却会大肆宣传来促销他们的产品。如果不懂消费性产品与工业产品之间的微妙差异，就很难成为赢家。

令人意外的是（或许结果不是那么令人意外），《Linux Journal》杂志（<http://www.linuxjournal.com/>）上可以找到一些在关键系统中应用Linux的最佳典范。我翻阅《Linux Journal》最近几年刊登过的文章，找到了几个基于Linux的非消费性嵌入式应用。这些内容，再结合Internet上找到的消费性电子产品细节，以及稍后将会看到的统计数据，可以客观地展现以Linux作为嵌入式操作系统的能耐及未来。表1-1包含了稍后将会讨论到的嵌入式Linux系统之范例的摘要。第一列简述范例系统，第二列用来说明嵌入式系统的类型，其余四列则是根据前一节提到的标准来描绘系统的特性。

表 1-1: 嵌入式 Linux 系统之范例的特性

说明	类型	规模	时限	网络能力	与用户交互的程度
加速器控制	工业的过程控制	中型	严格	有	低
计算机辅助训练系统	航空与航天	大型	严格	无	高
Ericsson "blip"	网络	小型	宽松	有	非常低
SCADA 协议转换器	工业的过程控制	中型	严格	无	非常低
Sharp Zaurus	消费性电子产品	中型	宽松	有	非常高
太空交通工具控制	航空与航天	大型	严格	有	高

加速器控制

加速器控制系统是欧洲同步辐射机构(European Synchrotron Radiation Facility, ESRF)建造的, 相关细节参见《Linux Journal》杂志第 66 期。加速装置由用来控制各种实验的许多硬件及软件组成。虽然并非所有软件都已经移植到 Linux 上, 不过一些令人感兴趣的部分都已经移植过去了。这包括串行线路及步进马达控制器。这些设备中已经有许多被用来控制各种系统。例如、串行线路用来控制真空装置、电源供给及可编程逻辑控制器(programmable logic controller, PLC)。此外、步进马达用来控制角度计(goniometer)、狭缝(slit)及移动平台(translation stage)。串行线路则是由 PC/104 上的串行板来控制。

用来控制串行板的 PC/104 单板计算机(single board computer, SBC)具备 Pentium 90 MHz 的 CPU 和 20 MB 的 RAM、以及 24 MB 的固态硬盘。一个标准的工作站发行套件, SuSE 5.3, 经过裁剪之后可以放进受限的永久性存储空间中。有些步进马达控制器运行在类似的配置之上, 有些则运行在具备 8~32MB 内存空间的 VME 板上、并且会使用 BOOTP/TFTP 从 Unix 类型的服务器下载操作系统(Richard Hirst 曾为基于 680x0 的 VME 板移植过 Linux、这些 VME 板所运行的操作系统便是修改自 Richard Hirst 移植的 Linux)。所有的设备都能够通过网络存取、并且可以通过 TCP/IP 网络控制。此处之所以会选用 Linux, 就重点来说, 是因为它的配置可以设定、稳定、免费、并且有良好的支持、支持许多标准, 以及可获得其源码。

计算机辅助训练系统

计算机辅助训练系统(computer-aided training system, CATS)是 CAE Electronics 公司开发的, 相关细节参见《Linux Journal》杂志第 64 期。与对视觉、声音和动作进行完整模拟的飞行仿真器不同, CATS 只会提供航空器上各种面板的视觉图像。然而, CATS 并非廉价的飞行仿真器, 它的目的在于提供初级训练, 以弥补飞行仿真器在这方

面的不足。CAE的CATS系统一般会构建在运行AIX的IBMRS/6000工作站上。之所以会移植到Linux上、是基于x86系统的物美价廉，以及Linux本身的可移植性这些方面的考虑。

CATS有三种不同的版本：一个、三个、七个屏幕的系统。开发及测试是在具有以下配备的工作站上完成：一个Pentium II 350 MHz的处理器、128 MB的RAM以及Color Graphics Systems的Evolution4图形卡（每张卡可控制4个显示器）。为了外接足够多的显示器，系统使用了Xi Graphics的AcceleratedX X服务器来控制Evolution4。不过，单屏幕的版本仍然可以轻易地在配备标准XFree86 X服务器的Linux系统上运行。

应客户的要求，该系统提供了可引导的CD-ROM，以避免在本地进行安装。所以只要使用CD-ROM和RAM文件系统就能运行整个CATS系统。这个系统最后的成效卓著，不仅在可靠性、稳定性上有良好的表现，在性能上也超过原有的要求。将飞行仿真器的雏形移植到Linux执行的工作始于2000年4月、而今成效卓著，目前大多数的全功能飞行仿真器也都移植到Linux上运行了。

Ericsson “blip”

Ericsson “blip”是一种商品。相关细节可以在Ericsson（爱立信）公司的blip网站（http://www.blipsystems.com/products_blipnet.shtml）和LinuxDevices.com网站上找到。blip是Bluetooth Local Infotainment Point（蓝牙本地娱乐信息存取点）的缩写，用来让蓝牙设备存取本地信息。此产品可在公共场所中提供服务，或是在家中进行本地信息的存取或同步。

这个blip产品包含基于ARM7TDMI的Atmel AT91F40816中央处理器（运行频率22.5MHz）、2 MB的RAM、1MB的系统闪存（system flash）和1MB的用户闪存（user flash）。这个Atmel芯片运行的是Lineo所发行的uClinux套件〔其内核版本为2.0.38，并针对无内存管理单元（MMU-less）的ARM芯片做过修改〕和uClibc（一个小型的C链接库），并且可通过串口跟独立的蓝牙芯片交互。该设备的存取功能来自一个专属的蓝牙堆栈、一个以太网网络接口，以及一个串口。如果要替blip开发定制应用，可以使用Ericsson提供的SDK及特制的GNU软件。选择Linux的理由，是因为它为主机与目标板提供了一个开放且便宜的开发环境，这样可以鼓励并促进第三方软件的开发。

SCADA 协议转换器

System Control and Data Acquisition（系统控制与数据获取，SCADA）协议转换器的相关细节请参考《Linux Journal》杂志第77期。其中提到，将原有在炼油厂中用来控制涡轮压缩机的数字控制系统（Digital Control System，DCS）整合进SCADA系统，

可以简化炼油厂的管理作业。尽管将整个DCS全都转换过去会得到较佳的整合性,但是代价太高,因此退而求其次、建立转换网关,以便兼容原有的DCS与SCADA系统。

之所以选用Linux,是因为它易于定制、有很好的说明文档、能够从RAM运行,以及能够直接在目标板上完成开发工作。存放应用程序的固态硬盘是M-Systems公司的8 MB DiskOnChip (DOC)。为了避免使用M-Systems提供的二进制驱动程序为内核打补丁,所以DOC的格式会维持出厂时的配置,即DOS文件系统格式(注2)。内核和根文件系统经过压缩之后,会被放入DOS格式的DOC中。引导时,批处理文件会调用Loadlin程序,以便加载内核及根文件系统。因此,系统文件仅供读取,并且以RAM来存储根文件系统。根文件系统是根据《BootDisk HOWTO》文件的说明、以Red Hat 6.1建立的。这个系统是一台具备32 MB RAM的工业级PC。

Sharp Zaurus

Sharp Zaurus是Sharp Electronics(夏普电器)销售的商品。Zaurus的相关细节请参考<http://www.myzaurus.com/>和LinuxDevices.com等网站上的说明。Zaurus是一台完全基于Linux的PDA,它配备的都是常见的PDA应用程序,如通讯簿、待办事项、日程表、记事本、计算器、电子邮件等等。

Zaurus最初的版本SL-5500采用Intel StrongARM 206 MHz处理器、64 MB的RAM和16 MB的flash。较新版本的SL-5600采用XScale 400 MHz处理器、32 MB的RAM和64 MB的flash。这个系统采用了Lineo公司的mbedix嵌入式Linux发行套件,以及QT公司的Palmtop GUI。为了鼓励Zaurus软件的独立开发,Sharp在<http://developer.sharpsec.com/>上维护了一个开发者网站。

太空交通工具控制

太空交通工具控制(space vehicle control)由欧洲太空总署(European Space Agency, ESA)开发,相关细节参见《Linux Journal》杂志第59期。自动运送运载工具(Automatic Transfer Vehicle, ATV)是个无人太空交通工具,用来填补燃料并且重新启动国际太空站(International Space Station, ISS)。ATV与ISS会合的过程中,ATV必须追上ISS,并且精确地对接。这个过程受到复杂的数学方程式的控制。由于太过复杂,监控系统必须确定所有的操作都照着计划进行。这个工作就是Ground Operator Assistant System(地面操作员辅助系统, GOAS)和Remote ATV Control at ISS(ISS端ATV控制, RACSI)的责任了。

注2: DOC不只有M-Systems的二进制驱动程序,还可以在第七章看到GPL的驱动程序。

GOAS 在地面执行，具备监控及干涉的能力，它的运行平台为：基于 UltraSPARC 5 的 Sun 工作站 (Solaris)、64 MB 的 RAM 及 300 MB 的磁盘空间。GOAS 现在已经移植到了 Linux，它的运行平台为：Pentium 233 MHz 及 48MB 的 RAM。

RACSI 在 ISS 端执行，用以提供临时性的任务中断及碰撞侦测，它的运行平台为：IBM ThinkPad、64 MB 的 RAM 以及有 40 MB 的磁盘空间可用。这个系统运行在 Slackware 3.0 发行套件之上，并且使用 Moo-Tiff 链接库以便提供 Motif 形式的窗口部件 (widget)。

之所以选用 Linux 是因为它具备了太空应用所需要的可靠、可移植、性能和可负担等要求。尽管有这些好处，但是基于操作上的理由，ESA 最后还是决定在 Solaris 上（使用相同的设备）执行 RACSI 和 GOAS。

如以上的范例所示，Linux 可以通过各种方式应用在各种领域，并且使用各种硬件及软件配置。采用 Linux 建立嵌入式系统时，最快的方法通常就是去参考一个已经在系统中使用 Linux 的类似计划。基于 Linux 的嵌入式系统范例还有更多是我没有探讨到的。附录二列出了各种资源，是一个很好的起点。但请记住，复制别人的计划也会连带复制别人的错误。最好的防范之道就是找出别人的问题，因此确定你了解系统的各个方面，或者至少要找到关于计划未知领域的更多信息。

调查的结果

自从 Linux 开始用作嵌入式操作系统之后，有许多调查报告出炉，它们提供的信息可以让我们了解 Linux 在这个领域应用的各方面情况。然而，这些调查的完整结果，有许多都是商业报告的一部分，价格相当昂贵，并且只公布了其中一些令人感兴趣的事实。现在让我们来看看这些调查结果。

2000 年的时候，《Embedded Systems Programming (ESP)》杂志对 547 位订户作了一项调查。这项调查发现，虽然在 1998 和 1999 年设计嵌入式系统的时候没有入会考虑到 Linux，但是有 38% 的读者表示，他们会考虑为下一次的设计使用 Linux 作为操作系统。令人特别感兴趣的是，Linux 排名仅次于 VxWorks（这是 WindRiver 公司的旗舰产品）。这项调查还发现，虽然在 1998 和 1999 年的时候没有人会在嵌入式系统上使用 Linux，但是到了 2000 年的时候，已经有 12% 的响应者在他们的嵌入式系统中使用了 Linux。

2000 到 2001 年期间，LinuxDevices.com 在网站上进行了一些调查（参见 <http://www.linuxdevices.com/cgi-bin/survey/survey.cgi?view=archive>），让网站访问者能够提供其在嵌入式系统中使用 Linux 的相关信息。在这两年间，收到了几百个访问者的回应。尽管没有过滤响应者的控制机制，但是调查的结果与其他较正式的调查却不谋而合。

LinuxDevices.com于2000年的调查发现,若考虑在嵌入式系统中使用Linux,大部分的开发者都打算在目标板上使用x86、ARM或PPC等CPU。这个调查显示,大部分开发者打算从DiskOnChip或系统自带的快闪设备来引导Linux,而且系统中主要的外围设备还要包括Ethernet和数据采集卡。开发者选用Linux的最重要理由,就是开放源码软件优于私有软件的地方,事实上有了源码不仅有助于理解操作系统,也可以避免依赖单一操作系统厂商的情况。这些开发者还表示,他们会选择Red Hat、Debian和MontaVista作为自己主要的嵌入式Linux发行套件。

LinuxDevices.com于2001年的调查发现,打算在嵌入式系统中使用Linux的开发者,其目标板多半会使用基于x86、ARM和PPC的系统。正如2000年的调查,大多数的开发者都打算以某些形式的flash存储设备来启动他们的系统。与2000年不同,2001年调查的目的是,看看开发者打算使用多少RAM和永久性存储设备。大多数开发者似乎都想将Linux使用在具备8 MB以上的RAM和永久性存储设备的系统。2001年的调查显示,开发者选用Linux的理由,是因为可获得Linux源码、Linux的可靠与稳固,以及Linux的模块化与可配置性。这些开发者还表示,他们会以Red Hat和Debian作为自己主要的嵌入式Linux发行套件。加上2000年的调查,LinuxDevices.com于2001年的调查结果证实了,开发者对Linux具有稳固的兴趣。

另一个对“在嵌入式系统中使用Linux”提出报告的组织是Venture Development Corporation(VDC)。尽管这份报告的调查对象是销售产品给嵌入式Linux开发者的公司,不过VDC于2001和2002年出版的报告中却透露了一些令人感兴趣的事实。首先,2001年的报告提到,嵌入式Linux开发工具的市场,2000年的产值是2千万美元,到了2005年预估产值为3亿6百万美元。2001年的报告还提到,这个市场的领导厂商包括Lineo、MontaVista和Red Hat。这份报告发现开发者选用Linux的主要理由包括:可获得源码、不用付版税。

VDC于2002年的报告包括了在网站上对11 000位开发者做调查。这个调查发现,目前开发者使用的Linux发行套件有Red Hat、Roll-Your-Own(译注4)以及非商业的发行套件。开发者选用Linux的主要理由包括:可获得源码、没有许可证问题、可靠,以及有开放源码社群的支持。值得注意的是,这份报告也列出了制约Linux在嵌入式系统中应用的主要因素,包括:实时的限制、对支持的可用性和质量持有怀疑,以及担心Linux存储空间不连续(fragmentation)。此外,这份报告还提到,开放源码社群对Linux相关技术问题的响应情况,大多数受访者表示他们都能获得了满意的答案。

Evans Data Corporation(EDC)也在2001和2002年提出了关于“在嵌入式系统中使用Linux”的调查报告。2001年调查过500位开发者之后,发现Linux在嵌入式系统目

译注4: 为特定硬件定制的发行套件。

前可能会用到的操作系统清单中，排第4位，而且Linux被认为是来年最被看好的嵌入式操作系统。2002年调查过444位开发者之后，发现Linux在嵌入式系统目前可能会用到的操作系统清单中，仍旧排第4位，而且Linux有可能和Windows一样在未来成为设计嵌入式系统时选用的操作系统。

以上所见只是部分的调查结果，就这样断言Linux会对嵌入式业界造成全面的影响仍嫌太早，不过显然开发者对嵌入式Linux系统已经有了很大的兴趣，而且这种兴趣与日俱增。这些调查结果还显示，开发者对Linux并非纯粹的业余爱好，他们认为Linux可用于专业的应用，并且优于许多传统的嵌入式操作系统。此外，专讲反话的FUD [试图造成恐惧 (fear)、不确定 (uncertainty) 和怀疑 (doubt)] 造谣说，Linux不好只因为它是免费的。事实上，Linux的好处多多：它的源码可自由获得、非常可靠，以及容易针对任务量身定制。值得注意的是，Debian发行套件仍然是受欢迎的嵌入式Linux之一，即使没有厂商在市场上销售该发行套件。

选用 Linux 的理由

除了以上各种调查所提到的理由之外，让开发者自愿舍传统嵌入式操作系统而取Linux，还包括各种原因。

程序代码的质量与可靠度

质量与可靠度是对程序代码有多少信心的主观度量。事实上，很难对优质的程序代码做出精确的定义，到底优质的程序代码应该具备哪些特性，下面仅列出一般程序设计者的看法：

模块化与结构化

不同的功能应该放在不同的模块，而且项目的文件目录层次应该反映该特性。在每个模块中，复杂的功能会被细分成适当数量的独立函数。

容易修改

凡是了解程序代码内部的人都应该能够轻易进行修改。

可扩充

轻易就可以将新功能加入程序。如果有需要变更程序的结构或逻辑，应该轻易就能找出要修改的地方。

可配置

可以选择程序代码的哪些功能应该出现在实际的应用中，而且可以轻易完成配置。

下面是可靠的程序代码应该具备的特性：

可预测

执行时，程序的行为应该要按照所定义的框架进行，不应该背离定义。

从错误中恢复的能力

在有问题发生时，希望程序会采取一些步骤，以便从问题中恢复过来，而且会使用有意义的诊断信息来警示有适当权限的人，通常就是系统管理员。

长期运行的能力

程序可以独自运行很长的一段时间，而且不管遇到什么情况，它都能够妥善保护自己。

多数程序设计者都一致同意，Linux 内核及 Linux 系统上大部分项目的程序代码库都符合以上对质量和可靠度所做的描述。理由是开放源码的开发模型（参见下面的提示）让许多人能够为开发计划撰写程序、找出既有的问题、讨论可能的解决方案，以及有效地修正问题。你可以预期，Linux 就算是在无人照顾的情况下，运行多年也不会出问题。你可以选择哪些是你想安装的系统组件，哪些不是。对内核来说，还可以选择在建立配置期间哪些是你想要的功能。程序代码的质量来自各个 Linux 组件，可以到各个邮件论坛看看：维护软件组件的每个人找出问题有多快，或加入新功能的速度有多快，这些都是最好的证明。很少有操作系统能够提供这种水准的质量和可靠度。

注意：严格地说，并没有此处所谓的“开放源码”开发模型，或甚至是“自由软件”开发模型。“开放源码”和“自由软件”其实只是一组许可证，用以保护藉此许可证发行的各式软件套件的版权。然而，这些藉“开放源码”和“自由软件”版权发行的软件套件通常遵循着类似的开发模型。Eric Raymond 在其所著由 O'Reilly 出版的《The Cathedral and the Bazaar》（大教堂与市集）中有关于这个开发模型的说明。

程序代码的可用性

你可以毫无限制地随意取用 Linux 的源码及所有生成工具，其程序代码的可用性由此可见一斑。最重要的 Linux 组件，包括内核本身，都是在 GNU 通用公共许可证（GNU General Public License, GPL）的保护下发行的。因此取用这些组件的源码是每个人的权利。其他组件则是在类似的许可证下发行的。这些许可证中有些，例如 BSD 许可证，则免除了二进制再发行时必须随附源码或修改后的源码的义务。然而，当你要取用构成 Linux 的各个开发计划的源码时，大部分都可以轻易获取，并且毫无限制。

当获取的源码有问题时,开放源码和自由软件社群就会以开放源码的版本提供类似的功能试图取代这种“有缺点的”软件。相对来说,无法获取传统嵌入式操作系统的源码,或是必须花大钱购买。获取源码的好处是,不需要外援,只要有能力钻研程序代码,看得懂它的运行原理,就可以自己修改程序。例如,一旦有人公布系统的问题,有能力的人通常就可以马上修正安全弱点或是性能瓶颈。使用传统的嵌入式系统,只能联络厂商,向他们报告系统的问题,并且等候他们修正问题。通常,人们宁可自己寻求解决方案,也不愿等待别人修正问题。大型计划的负责人甚至会花钱购买源码,以降低对外界的依赖。

对硬件的支持

广泛的硬件支持,意味着Linux支持各种不同类型的硬件平台与设备。尽管仍有些厂商尚不提供Linux驱动程序,但预计会有越来越多的厂商加入提供Linux驱动程序的阵容。因为有大量的驱动程序是靠Linux社群自己维护的,所以你大可放心地使用这些硬件组件,而不必担心有朝一日厂商会停止该生产线。广泛的硬件支持,同时意味着Linux可以运行在各种不同的硬件平台上。同样地,没有其他的操作系统可以提供这种水准的可移植性。随便指出一种CPU或平台,都能够合理地推测Linux可以在其上运行,或是已经有人做过类似的移植程序,而且可以帮助你。你还能够推测你在某个架构撰写的软件,可以轻易移植到Linux运行的另一个架构上。有些设备驱动程序甚至不需要修改就能在不同的硬件架构上运行。

通信协议与软件标准

正如我们将在本书中看到的,Linux还提供广泛的通信协议及软件标准支持。这让我们可以轻易地将Linux整合进既有架构,以及将过去的软件移植到Linux上。你可以轻易地将Linux系统整合进既有的Windows网络,并预期它可以通过Samba为各个客户端提供服务,不过客户端看到的会与在NT服务器上看到的有些差异。如果打算利用Linux机器进行业余无线电通信,只需在内核加入此功能即可。同样地,Linux是一种Unix克隆(即与Unix兼容的操作系统),可以轻易地将传统的Unix程序移植到Linux上。事实上,现在所发行的各种应用程序,只要最初是在商业Unix上运行的,稍后都会移植到Linux上。这包括了FSF提供的所有软件。大多数传统的嵌入式操作系统,就这一点来说限制非常大,厂商通常只对协议和软件标准的子集提供有限的支持。

可用工具

目前Linux有各式各样的工具可用,这让Linux成为功能强大的操作系统。如果需要使

个工具写了出来，并且放在 Internet 上供人随意取用。这就是 Linus Torvalds 为众人所做的事。请访问 Freshmeat (<http://www.freshmeat.net>) 和 SourceForge (<http://www.sourceforge.net>) 等网站，看看有哪些工具可用。

社群 (Community) 的支持

受到社群的支持或许是 Linux 最大的优点。这就是自由软件及开放源码的精神所在。当需要使用某个应用程序时，很可能早在你之前就有人遇到跟你类似的情况。通常，这个人将会很乐意跟你分享他的解决方案，为你回答问题。与开发和支持有关的邮件论坛，是找到这类社群支持的最佳场所，在这些地方找到的专家支持的级别通常胜过私有操作系统厂商昂贵的电话支持。通常，当你打技术支持电话时，根本不可能让你直接询问开发该软件的工程师。使用 Linux 时，只要将一封电子邮件寄往正确的邮件论坛，通常就可以直接联络开发软件的那个人。之后就是非常流畅的交互过程：指出漏洞、获得修正码或建议。有许多程序设计者会遇到没人理会的窘境，他们会被要求搜寻已归档的邮件，以确定他们的问题尚未被回答过。

许可

Linux 的许可让程序设计者能够为 Linux 做他们无法为私有软件做的事。基本上，可以使用、修改并再次发行该软件，但仅限于向该软件的用户提供相同的权利。这就是 Linux 用到的各种许可证 (GPL、LGPL、BSD、MPL 等等) 的主要精神。但这并不代表你会就此丧失你所创作的软件中体现的版权和专利权。相关问题将会在“版权和专利权”一节讨论到。这样的许可为 Linux 的可用性提供了相当大的自由度。

不依赖特定厂商

不依赖特定厂商，意味着要获取或使用 Linux 不需要依赖单一厂商。此外，只要不满意某个厂商的服务态度，大可以更换厂商，因为 Linux 发行时的许可证，让你即使更换厂商也能获得相同的权利。不过，有些厂商会在自己的发行套件中额外提供非开放源码的软件，在你更换厂商之后，可能就无法使用这类软件了。当你要选用任何发行套件时，必须考虑到这个问题。你可以将 Linux 比拟成一辆车，因为引擎盖并未焊死，可以自己找技工提供服务，不必像私有软件那样必须找原经销商的技工提供服务。

成本

Linux 的成本是开放源码许可的结果，这不同于传统的嵌入式操作系统。建立传统嵌入式系统有三种软件组件成本：基本开发套件、额外的工具、运行时的版税。基本开发套件的费用包括向操作系统厂商购买开发许可证。通常必须购买可供一定人数使用的许可

证，每位开发者都必须获得许可证。此外，可能会发现基本开发套件提供的工具不够用，想要向厂商购买额外的工具。这是另一项花费。最后，当你要部署系统的时候，厂商会向你索要每个设备的版税。整个成本的多少取决于产品类型以及产量。例如，有些移动电话的制造商会选择自己开发操作系统，以便避免付版税。这一点对他们很重要，因为销售量与利润有关。

使用Linux的成本模型又是另一回事了。所有开发工具和操作系统组件都可以免费获得，而且它发行时的许可证是为了避免有人对内核组件收取任何版税。然而，大多数的开发者可能都不会想要自己寻找各种软件工具和组件，以及分析哪些版本兼容哪些不兼容。大多数的开发者宁可使用已经包装好的发行套件。这包括购买发行套件，或是自己下载。在这种情况下，客户需要付费并且可能会要求发行套件厂商提供支持，或是要求厂商将他们的发行套件移植到新的架构，以及开发新的驱动程序。这就是发行套件厂商能够赚钱的地方。他们还会收取发行套件中附带的私有软件的费用。与传统嵌入式软件的成本模型比起来，这已经是相当低的成本了，实际的成本跟你所选用的发行套件有关。

嵌入式Linux舞台的参与者

与私有的操作系统不同，Linux并没有让任何单一的权力实体支配它的未来、设计哲学以及要采取何种技术。这些问题都是由一群使命和目标不同但功能互补的参与者共同处理的。

自由软件和开放源码社群

自由软件和开放源码社群是Linux发展的原动力，也是嵌入式Linux竞技场上最重要的参与者。它们的成员全都是为Linux系统的各个软件组件增强功能、进行维护及提供支持的开发者。它们没有领导中心，各自独立，各有各的专长。你可以在与它们有关的邮件论坛上或集会场所中（如Ottawa Linux Symposium）看到它们的成员进行技术问题的讨论。我实在很难将它们归类在一起，因为它们的背景不同，成员也不同。但是，它们都非常在意自己开发的软件的技术质量。Linux的质量与可靠度，正如之前所论述的，就是它们这种在意程度的体现。

我本身是自由软件社群的成员，并且做过一些贡献。除了维护一些现有的邮件论坛，还以各种形式参与自由软件的推动，并负责撰写及维护Linux Trace Toolkit（这是一组用来跟踪Linux内核的组件）。我也对其他自由软件和开放源码计划，包括RTAI和Adeos，做出过贡献。

我在本书中将会提到相当多Linux系统用到的组件。这些组件的维护者或贡献者，也都是自由软件和开放源码社群的参与者。

业界

由于Linux在嵌入式市场上有发展潜力，促使许多公司开始在这个领域接纳和推广Linux。业界的参与之所以重要，是因为他们会将Linux做成消费性产品。通常，他们会最早收到来自用户的回馈。尽管开发者可以在各种邮件论坛看到产品的使用情况，但并非所有用户都会参与邮件论坛的讨论。因此厂商必须同时照顾到一般用户和各种Linux计划的开发者，以免陷入到头来必须自己开发的窘境。就这一点来说，许多厂商在嵌入式Linux市场上的定位都很成功。以下仅列出其中部分厂商。

注意： 以下所列出的厂商只是基于讨论的目的。作者并未对这些厂商所供的服务做出任何价值判断，因此这份清单不应该被解读成任何形式的背书。

Red Hat

如果说它不是最常用的Linux发行套件，它也是最常用的Linux发行套件之一。其他发行套件的灵感都来自该套件，或至少要考虑到它。Red Hat是最早期的Linux发行套件之一，Red Hat取之社群又贡献回社群的做法，让它建立起领导品牌的地位。并购Cygnus之后，Red Hat旗下增添了一些GNU开发工具链的主要开发者。主要的Linux贡献者中又有一批人为Red Hat工作。Cygnus在被并购之前的业务是为嵌入式系统开发者提供商业版的开发工具套件。虽然Red Hat并购Cygnus之后仍持续此项业务，但却是以GPL的版权发行。详情参见<http://sources.redhat.com/>。

MontaVista

MontaVista的创办人是嵌入式工业的老兵Jim Ready。MontaVista的产品、服务以及对Linux在工业应用的推动上都被公认为嵌入式Linux市场的领导品牌。它的主要产品是MontaVista Linux，这个产品有两种版本：Professional和Carrier Grade（译注5）。MontaVista对开放源码计划的贡献包括：内核、ViewML、Microwindows和LTT（Linux Trace Toolkit）。虽然MontaVista并未替其所贡献的项目维护网站，不过你可以在<http://www.mvista.com/developer/sourceforge.html>上找到相关的链接。

LinuxWorks

就是大家熟知的Lynx Real-Time Systems公司，它是传统的嵌入式操作系统厂商。与其他传统的嵌入式操作系统供货商不同的是，Lynx很早就决定容纳Linux，并且

译注5：在编写本书时，当时又多了一种版本Consumer Electronics。

为公司更名以明其志（译注6）。再加上后来 WindRiver 公司并购 BSDi 的操作系统部门（注3）以及 QNX 公司决定提供操作系统供人自由下载等事件所展现的开放源码趋势，尤其是 Linux 对嵌入式竞技场的严重侵犯。然而 LynuxWorks 并未放弃对 Lynx 的开发、发行及支持。事实上，LynuxWorks 推销的是双重解决方案。根据 LynuxWorks 的说法，需要硬实时性能的程序设计者应该继续使用 Lynx，而想要使用开放源码解决方案的人，则应该使用 BlueCat（这是他们的嵌入式 Linux 发行套件）。LynuxWorks 甚至修改了 Lynx，使得 Linux 的二进制代码不需要重新编译就能在 Lynx 上执行。事实上，LynuxWorks 早就是成功的嵌入式操作系统厂商了，连它都这么早就采用 Linux，这证明了在嵌入式市场中迈向开放源码操作系统的重要性。

还有许多规模较小的参与者为开放源码和自由软件提供了各种服务。事实上，开放源码和自由软件中有许多贡献来自独立的或为小厂商工作的个人。此外，这些规模较小的参与者所提供的服务通常不亚于而且有时还会优于规模较大的参与者。下面所列举的都是为嵌入式 Linux 提供服务的个人及小公司，他们同时是开放源码和自由软件社群中活跃的贡献者：Alessandro Rubini、Bill Gatliff、CodePoet Consulting、DENX Software Engineering、Opersys、Pengutronix、System Design & Consulting Services 以及 Zee2。

各种组织

目前有三个组织致力于推动及促进在嵌入式应用中采用 Linux：Embedded Linux Consortium（嵌入式 Linux 联盟，ELC）、Emblix（Japan Embedded Linux Consortium）以及 TV Linux alliance。ELC 最初由 23 家公司组成，它是一个不偏向任何厂商的非营利组织，现在其成员超过 100 家公司。它受 Linux Standard Base 及 Single Unix Specification 的启发，现在的目标是为嵌入式 Linux 建立平台规范。开放源码和自由软件社群中与此规范有关的开发者是否会接纳 ELC 的规范仍不明确，因为这个标准的草案并未开放，这有违开放源码和自由软件的传统。Emblix 最初由 24 家日本公司组成，它的目标与 ELC 类似，只不过重点放在日本市场。TV Linux alliance 的成员包括有线、卫星及电信技术的提供商及运营商，他们的共同目标是想在机顶盒和交互电视应用上支持 Linux。

除了以上这三个值得注意的组织，在各领域里还有对 Linux 的发展会造成影响的其他团体，只不过他们并未特别针对嵌入式系统。

译注 6：Lynux=Lynx+Linux。

注 3：后来 WindRiver 改变心意，它与 BSD 的关系似乎只是过去的事。

其中，最首要的是 Richard Stallman（GNU 计划的维护者，Linux 系统的基本组件大多来自 GNU 计划）于 1984 年发起的自由软件基金会（Free Software Foundation, FSF）。FSF 也是 GPL 和 LGPL 版权的授权中心，它所授权的软件大多为 Linux 系统吸纳。FSF 的宗旨就是在与计算机有关的各个领域提供自由软件（free software）（注 4）让大众使用。FSF 注意到 GNU 和 GPL 版权的软件在嵌入式系统中的使用近来有增加的趋势，于是开始将目光移往此处，希望能确保用户与开发者的权利。

OpenGroup 维护的 Single Unix Specification (SUS) 是在描述一个 Unix 系统应该包含哪些东西。这正是 Linux Standard Base (LSB) 努力的目标，也就是 LSB 在网站 (<http://www.linuxbase.org/>) 上开宗明义所说的“开发与推动一套标准，这将能提升各 Linux 套件的兼容性，让应用程序能够在任何兼容的 Linux 系统上执行”这句话。此外，Filesystem Hierarchy Standard Group 维护的 Filesystem Hierarchy Standard (FHS) 标准是在规范 Linux 的根文件系统的内容。Free Standards Group (FSG) 维护的 Linux Development Platform Specification (LDPS) 则在规定开发平台的配置，只要是在符合规范的平台开发的应用程序，就能在大多数的发行套件上执行。最后值得一提的是 Real-Time Linux Foundation，它的目标是推动及标准化 Linux 的实时应用和程序设计。

资源

多数开发者都是通过各种资源网站和出版物进入嵌入式 Linux 世界的。开发 Linux 的社群、业界及组织则是通过这些网站和出版物来公布其成果，并得知其他参与者的成果。本质上，这些资源网站和出版物就是所有关心嵌入式 Linux 发展的人聚会的场所。附录二中列出了一份资源清单，其中以 LinuxDevices.com 网站和《Linux Journal》杂志最为突出。

LinuxDevices.com 由 Rick Lehrbaum 于 1999 年万圣节（注 5）创立。LinuxDevices.com

注 4： 这里的“free”就是“free speech”（言论自由）里的“free”，指的是“自由”的意思，而非“free beer”（免费啤酒）里的“free”。Richard Stallman 注意到这是英语所造成的混淆，其他语言就不会造成困扰，像是法语的“libre”（自由）与“gratuit”（免费）就是使用不同的词。英语的“free software”（自由软件）可以正确地翻译成法语的“logiciel libre”。

注 5： 之所以会选这个日子，是为了纪念 Eric Raymond 所披露之恶名满天下的《万圣节文件》（Halloween Documents）（译注 7）。细节请参考 <http://www.opensource.org/halloween/>。

译注 7： 微软内部一份机密的备忘录（内容为用来对付 Linux 和开放源码的策略）在 1998 年 10 月份的最后一周泄漏出来，Eric Raymond 予以揭露，希望世人认清微软的所作所为。

之后被 ZDNet 并购，后来又被 Rick 拥有的公司买回。至今，仍是由 Rick 在维护这个网站。LinuxDevices.com 的特色包括：新闻、文章、投票、论坛以及许多与嵌入式 Linux 有关的链接。许多与嵌入式 Linux 有关的重要公告都是在这个网站发布的，此处也可以找到许多与嵌入式 Linux 有关的文章。尽管 LinuxDevices.com 有浓厚的商业气息，我还是建议各位偶尔浏览一下此网站，以便了解嵌入式 Linux 的发展近况。此外，LinuxDevices.com 对 Embedded Linux Consortium 的创立也有很大的帮助。

有鉴于 Linux 在嵌入式系统方面的应用越来越重要，《Linux Journal》(LJ) 杂志的发行公司 Specialized System Consultants (SSC) 于是在 2001 年 1 月出版了《Embedded Linux Journal》(ELJ) 杂志，以便服务嵌入式 Linux 社群，不过后来停刊了。尽管 ELJ 不再作为独立的杂志发行，不过 LJ 现在每个月都会包含 embedded 方面的文章，这些文章都是新的，并非是曾在 ELJ 上发表过的文章。

版权和专利权

你或许会问：在我的设计中使用 Linux 会怎样？用来保护 Linux 的诡异许可证，是否会危害到我公司的版权和专利权？这些许可证到底在说什么？是否每种许可证我都要详细阅读？我可以发行二进制形式的内核模块吗？为什么有些评论文章甚至说 Linux 的许可证是一种“病毒”？

在你的心中或许还萦绕着更多问题。你甚至可能已经跟某些同事讨论过这方面的问题。这些问题如果不妥善处理，可能会越搞越迷糊。这些问题的确存在，要避免由于使用 Linux 而造成某种许可污染的问题，其实是有办法的。有一件事很重要，请铭记在心，接下来的说明与法律无关，而且我也不是律师。如果对自己的计划有任何疑问，最好去请教律师。

好吧，既然要为你详细讲解，就让我们来看看大家对 Linux 的许可证的共同见解，以及通常都是怎样将它应用在 Linux 系统上的，包括嵌入式系统。

GPL 的典型应用

组成 Linux 系统的大多数组件将会受到两种先前介绍过的许可证——GPL 和 LGPL——的保护。这两种许可证的文件可以从 FSF 位于 <http://www.gnu.org/licenses/> 的网站获得，而且应该包含在受到这些许可证保护的任何发行套件中（注 6）。GPL 主要的保护

注 6： GPL 用来放许可证的文件通常称为 *COPYING*；LGPL 则称为 *COPYING.LIB*。当发行套件安装好之后，这些文件可能会被安置在磁盘的某处。

对象是应用程序，LGPL主要的保护对象则是链接库。凡是内核、二进制形式的工具程序、gcc编译器、gdb调试器都在GPL的保护之列。另一方面，C链接库和GTK widget toolkit则受到LGPL的保护。

有些程序可能受到BSD、Mozilla或其他许可证的保护，不过GPL和LGPL是主流。无论使用哪种许可证，该有的常识不能没有。确定你了解自己使用的组件受到哪些许可证的保护，以及它们的真正含意。

GPL提供的权利与义务与典型的软件许可证不同。本质上，GPL的用意在于为开发者与用户提供较高的自由度，让他们能够在几乎没有限制的情况下使用、修改并发布软件。为了确保这些权利不会受到任何形式的取消或劫夺，GPL还做了以下约定：

- 只要不更改许可证和版权声明，可以随意复制程序。
- GPL对软件提供的许可不包含任何担保，除非该担保是由发行者提供的。
- 为别人提供复制品及担保服务时，可以收取费用。
- 发行程序的二进制副本时，必须随附源码，这通常是指“本来的”源码（注7）。
- 再次发行软件，不能进一步限制获得该软件的人、GPL和该软件原始作者授予的权利。
- 你可以修改程序并且发行你改过的程序，但你必须给获得该程序的人提供相同的权利。实际上，任何程序代码只要修改或包含了受GPL保护的程序代码，或是GPL程序的任何一部分，若不遵照GPL的规定，就无法在公司以外发行。这也难怪有些公关人员会指责GPL就像病毒一般。但请记住，此限制仅限于源码。正如稍后所见，如果为了执行的目的，将未经修改的软件包装起来，则不在此限。

正如所见，GPL不仅提供取用软件的自由，也保护作者的版权。这是毫无疑问的。然而，条款中提到应用程序的修改与发行，却产生极大的混乱。为了拨乱反正，必须就两点来说明：执行GPL软件、修改GPL软件。执行软件的行为通常就是原始作者撰写软件的用意。例如，撰写gcc的作者，就是要用它来编译软件。一个软件如果是由未经修改的gcc编译的，则不受GPL的制约，因为编译程序的动作只是在执行gcc。事实上，可以用gcc来编译私有软件，人们已经这么做好多年了，而且不必担心受到GPL的“污染”。相对来说，修改软件的行为就是在原有软件的基础上建立衍生作品，因此就会受到原有

注7： GPL在授权条款中特别指出：“作品的原始码必须是最适合用来修改作品的形式。”以前有人在发行二进制的作品时，为了规避GPL，不提供本来的原始码，反而提供内容经过混淆的版本，这是违法的行为。

软件许可条款的制约。举例来说，如果在获得 gcc 编译器之后，将 gcc 修改成能够编译新的程序语言，新编译器就是一个衍生作品，因此你所做的任何修改，若不遵照 GPL 的规定，就无法对外发行。

一些反对 GPL 的言论或文章，多数是在 GPL 软件的执行和修改上打模糊战，刻意让人们有一个印象：任何软件只要跟 GPL 软件沾上边，就会有受到 GPL “污染” 的疑虑。事实并非如此。

软件的执行与修改显然不同。身为一个开发者，只需扪心自问，要执行既有软件（倘若这样就可以用的话），还是要根据需要修改软件，就可以免除任何麻烦。身为一个开发者，应该有相当的能力做出判断。

请注意，版权法并不会区分静态与动态链接。即使私有应用程序是在执行期间通过动态链接整合进 GPL 软件，仍然要受到 GPL 的制约。一个结合 GPL 软件的衍生作品，以及非 GPL 软件但与 GPL 软件有任何形式的链接关系，若不遵照 GPL 的规定，就无法对外发行。如果将 gcc 包装成动态链接库，并且使用该链接库来撰写新的编译器，那么新编译器仍然受到 GPL 的制约，若不遵照 GPL 的规定，就无法对外发行。

鉴于你只要在自己的程序中引用部分的 GPL 程序，就必须受到 GPL 的制约，否则不得对外发行，这点太过严苛，LGPL 允许你在自己的程序中使用部分未经修改的 LGPL 程序，不会有任何问题。但是，如果修改了 LGPL 程序，就像 GPL 一样，若不遵照 LGPL 的规定，就无法对外发行。若只是将私有应用程序以动态或静态的方式链接至受到 LGPL 保护的 C 链接库，则毫无限制。另一方面，如果修改了 C 链接库，则你所做的任何变更，必须受到 LGPL 的制约，否则无法对外发行。

注意： 请注意，如果所发行的私有软件有链接至 LGPL 软件，则必须考虑 LGPL 软件的代换问题。举例来说，如果将应用程序动态链接至一个链接库，则代换的问题很好解决，因为取用你软件的人只需变更应用程序在启动时所链接的链接库。然而，如果将应用程序静态链接至 LGPL 软件，则还必须给取用软件的人提供应用程序在链接之前的目标码，如此便能代换 LGPL 软件。

如同前面讨论的执行与修改 GPL 软件的差别，链接与修改 LGPL 软件也有明显的区别。你可以使用任何的许可证来发行软件，尽管该软件链接至 LGPL 链接库。但是，如果对 LGPL 链接库做了任何的修改，则必须遵照 LGPL 的规定，否则不得对外发行你所做的任何变更。

一些有待探讨的问题

到目前为止，我只探讨了GPL和LGPL的典型应用。一些其他的应用尚未明确加以定义。应用程序执行时若用到Linux内核会怎样？如果应用程序与内核的程序代码没有链接关系会怎样？与内核的整合更紧密的二进制内核模块会怎样？它们会受到GPL的制约吗？在嵌入式系统中引用GPL软件又会怎样？

我将会从最后一个问题开始探讨。嵌入式系统中若包含GPL软件，这实际上就是一个典型的GPL应用。不要忘了，如果想要发行GPL软件的二进制副本，必须让取用它的人也能够同时获得本来的源码。发行嵌入式系统使用的GPL软件，就是发行二进制的一种形式，只要你能够遵守GPL中跟执行和修改GPL软件有关的其他条款，这是允许的。

有些私有软件厂商对“在嵌入式系统中使用GPL软件”提出疑义，他们宣称以嵌入式系统中各应用程序间的耦合程度来看，根本就无法区分哪些属于GPL，哪些不属于GPL。但这并非事实，正如你将在第六和第八章看到的，嵌入式Linux系统的发行套件有办法做到模块化，并且能够分成不同的软件组件。

为避免在Linux内核上执行用户应用程序造成的混淆，Linus Torvalds在内核源码的GPL许可中加了一段前言。这段前言原文照登在附录三中，它规定凡是在内核上执行的用户应用程序都不受GPL的制约。也就是说，可以在Linux内核上执行任何类型的应用程序，而且不用担心GPL“污染”的问题。现在已经有许多厂商，包括Oracle、IBM和Adobe，提供可以在Linux上执行的用户应用程序，然而这些应用程序仍属于私有软件。

还有一件事不是十分明确，那就是纯二进制的内核模块。这些模块就是可以动态加载及卸载的软件组件，可以为内核加入新的功能。尽管它们主要的用途是设备驱动程序，不过也可以用于其他目的。内核的许多软件组件实际上都可以构建成可加载的模块，以降低内核映像的大小。需要时，内核可以在执行期间加载各种模块。

尽管使用模块是为了能够方便进行定制硬件架构，但许多厂商和项目在利用模块为内核提供功能的同时也保留了源码的控制权，或是以不同于GPL的许可方式发行模块。例如，有些硬件制造商会不向用户提供源码，只提供纯二进制模块驱动程序。这使得Linux用户在使用这些硬件的时候，无法要求厂商提供其设备的操作细节。

问题是，一旦模块被加载进内核，模块实际上就会变成内核地址空间的一部分，并且与内核有极紧密的耦合关系，因为模块调用的功能及获得的服务都来自内核。由于内核本身受到GPL的制约，有些人主张，模块的发行也必须受到GPL的制约，因为这样产生的内核就是一个衍生作品。有些人主张，应该允许发行纯二进制的模块，只要它用的是内核提供给模块使用的标准服务。对本身就是GPL软件的模块来说，这个问题并无实际

意义，但对非 GPL 软件的模块来说，这个问题就非常重要了。Linus 再三强调，只要能够证明该模块所实现的并非 Linux 特有的功能，就允许发行纯二进制模块。然而，正如你在附录三中看到的 Linus 的某些信件，有些人（包括 Alan Cox）质疑 Linus 能否代表大家做这个决定，因为 Linus 并未拥有所有内核程序代码的版权。有些人仍然主张，就是因为容忍纯二进制模块太久了，它们俨然已成为实际标准的一部分。

即使未用到内核 API，也是属于纯二进制模块的应用。RTAI 和 RTLinux 将实时任务插入内核就是个好例子。尽管有人可能会争辩说，这些模块并未用到内核 API，应该以不同的方式处理，但是它们仍然会被链接到内核空间，所以仍需当成一般模块来处理，无论你是否这么想。

在写作本书时，纯二进制模块问题还是没有明确、公认的处理方式，尽管它们仍被广泛应用并且被认为是合法。Linus 在最近一次对此事发表的公开言论中提到，从大家在内核邮件论坛对 Linux Security Module 基础建设的争辩看来（原文照登在附录三），呈现出一个事实，使用纯二进制模块将会是一项冒险的决定。事实上，使用纯二进制模块在可见的未来将会导致法律上的疑义。如果觉得需要在系统上使用纯二进制的私有模块，我建议你接受 Alan Cox 的忠告，预先找好律师以备不时之需。实际上，我还建议你重新考虑，是否要换成受 GPL 保护的模块。这将会免除你许多麻烦。

RTLinux 的专利权

或许当你为嵌入式系统部署 Linux 时，将会遭遇到的最大限制和许可争议之一，就是 RTLinux 计划的主持人 Victor Yodaiken 持有 RTLinux 的专利权。该专利涵盖 RTLinux 实现的对通用操作系统附加的实时支持。

尽管有许多人质疑该项专利的可行性、这是以前就有的技术、Victor 的处理态度等等，Victor 目前的确合法持有该项专利和许可证，至少在美国你不能视而不见。RTLinux 专利的美国专利编号是 5,995,745，而你可以通过适当的渠道获取其副本（译注 8）。你可以从 http://www.fsmlabs.com/products/rtlinuxpro/rtlinux_patent.html 获得使用该项专利技术的许可证。

这份许可证列出了免费使用该项专利技术的几个条件。特别是，许可证中有认可两种使用专利技术的方式。第一种方式就是使用 GPL 软件，第二种方式就是将软件使用在 Victor Yodaiken 的公司 FSMLabs 发行的 Open RTLinux 未经修改的版本上。若想采用传统的方式，实时 Linux 应用的开发者必须遵守以下规定：任何人只要发行非 GPL 的实

译注 8：可通过 http://ep.espacenet.com/espacenet/ep/en/e_net.htm。

时应用，都必须向 FSMLabs 购买许可证。FSF 的首席律师 Eben Moglen 并不这样认为。在一封寄给 RTAI 社群的信上（原文可从 <http://www.aero.polimi.it/~rtai/documentation/articles/moglen.html> 获得），Moglen 做了如下的陈述：“在运行中的 RTLinux 或 RTAI 系统上，并没有这样的应用会做到该专利所宣称的事。因此没有任何应用程序有可能会侵犯到专利权，也就是说没有任何程序必须或需要获得许可。”

尽管 Moglen 权威性的陈述相当明确，但是 FSMLabs 仍然拒绝说明该项专利的适用范围，这让使用其专利技术的所有实时扩充的许可问题变得疑云重重。

想要拨云见日只有避免使用专利技术一种办法。换言之，要让 Linux 支持实时响应的功能，必须使用专利未提到的方法。还好真的存在这个方法。

nanokernel（奈内核）的科学论文至少比 RTLinux 最初的专利应用（专利生效日为 1999 年 11 月 30 日）还早一年问世，我曾据此撰写了一篇白皮书，说明如何实现基于 Linux 的 nanokernel，让多个操作系统共享相同的硬件。这篇白皮书于 2001 年 2 月发表，标题为“Adaptive Domain Environment for Operating Systems”，可以从 <http://www.opersys.com/adeos/> 获得，在这里也可以看到其他文件说明此技术的其他可能用法。但我必须承认的是，这篇文档发表后不久，我就开始本书的撰写工作了，已经没有什么时间可以投入计划的开发，让这个构想沉寂了一年。

直到 2002 年 4 月底，一位很有天赋的自由软件开发者 Philippe Gerum 愿意接手并推动此计划。在 6 月初的时候，我们对计划的进展感到非常满意，于是决定对外发表 Adeos nanokernel。发表日期定于 2002 年 6 月 3 日，全世界数个自由软件组织，包括 Eurolinux (<http://www.eurolinux.org/>) 和 April (<http://www.april.org>) 在内，都认为这是一个不涉及专利权并且能让实时内核与通用内核共存的方法。尽管就像任何其他的专利一样，这样的背书并不能保证不会违反专利宣称的权利，但是开放源码和自由软件社群仍一致认为 Adeos nanokernel 及其应用的确不涉及专利权。至于我则鼓励各位自己去确认，这正是对任何专利应该持有的处理态度。此外，也不要忘了检阅最初的白皮书，最重要的是其中提到的科学论文。

现在全世界的开发者都已经在使用 Adeos，让不同类型的内核能够共存。例如，RTAI 之前便是使用专利技术来控制 Linux，因此受到专利权的制约，不过现在已经移植到 Adeos。尽管撰写本书时 Adeos 只能在单一处理器或 SMP x86 系统上执行，由于 nanokernel 相当简单，要移植到其他架构应该不会太难。如果想要为 Adeos 贡献心力，例如将它移植到其他架构，或者你只想使用它或获得更多信息，可访问该计划位于 <http://www.adeos.org/> 的网站。

使用发行套件

使用发行套件不是比自己从头开始设置开发环境及构建整个目标系统还要简单迅速吗？哪种发行套件最好？可惜，这些问题并没有确定的答案。不过，接下来要探讨的、有关发行套件的使用问题，或许能够帮助你找到这些问题及类似问题的答案。

用与不用

首先，应该认清的是，要构建嵌入式 Linux 系统并不需要使用任何形式的发行套件。事实上，所有必要的软件套件都可以在 Internet 下载得到。发行套件供应者也会下载相同的套件，并且包装起来供你使用。这么做可让你高度掌控并了解自己所使用的套件以及套件间的交互关系。尽管这是最周密的做法以及本书呈现的方式，但也最旷日废时，因为你必须花时间找到版本相符的套件，然后逐一设置每个套件，以便符合套件间的交互关系。

因此，如果需要高度掌控系统的内容，“自己动手做”或许是最佳的方式。然而，如果像大多数人那样，需要计划早就准备妥当，或者不想自己维护套件，应该认真考虑使用开发及目标板发行套件。如果真是这样的话，将需要选择最适合自己使用的开发及目标板发行套件。

如何选用发行套件

选用发行套件的时候，有一些标准可协助你做判断，前面已经述及其中一部分。但因计划的不同，可能还会有其他需要用到的判断标准尚未在此处讨论。不管怎样，如果选用的是商业发行套件，当你要评估产品的时候，确定发行套件厂商能够明确地回答问题。在任何情况下，如果问得空泛，将会得到空泛的回答；如果问得详细，理应得到详细的回答。精确的问题若得到不明确的答案，通常意味着有些不对头。然而，如果选用的是开放源码的发行套件（注8），设法让你所获得的相关信息越多越好。选用开放源码发行套件与商业发行套件的差别在于，获得答案的方式不同。由于商业发行套件厂商只会针对自己的产品来回答问题，所以如果你使用的开放源码发行套件有同样的问题，可能必须自己找答案。

选用开发或目标板发行套件时一开始会考虑到的是许可或者与许可相关的问题。有些商业发行套件包含部分的开放源码，并且在传统的软件许可声明下提供增值套件，所以禁

注8： 开放源码发行套件的维护者是开放源码社群，比方说Debian。这类发行套件本来就不会包含任何私有软件。

止复制而且要抽版税。确定发行套件的许可证有明确陈述增值软件的法及其适用范围。如果不明确，就去问个清楚。一定要搞清楚许可问题。

在评估发行套件之前，请先确定其中找得到你想要使用的套件。发行套件可能会提供更好的东西，不过你至少知道它是否符合基本需求。一个开发发行套件应该包含后面“开发工具的设置与使用”一节中提到的条目。目标板发行套件，就某个程度上来说，应该自动或很容易就会包含“建立目标板 Linux 系统”和“网络功能”两节中提到的条目。当然，没有一个发行套件可以排除“为嵌入式系统开发软件”一节中探讨的问题，即使只有系统开发者知道，何种形式的程序设计风格才符合系统的需要。

有一件事可以区分商业发行套件与开放源码发行套件，那就是厂商所提供的支持。商业发行套件厂商几乎总是支持自己的产品，开放源码社群对开放源码发行套件则不需要提供商业厂商同等级的支持。不过，有些厂商会对开放源码发行套件提供商业等级的支持。在嵌入式领域，通过服务不同客户的不同需要，各厂商对所支持的发行套件都会获得独特的知识，以及知道客户在使用发行套件期间可能遇到哪些问题，因此对你而言他们是你获得有效帮助的最佳处所。然而，这些厂商主要的工作还是跟上 Linux 版本发展的脚步，因此他们是你了解可能的漏洞和互操作性问题的最佳数据来源。

选用发行套件的时候，还可能会受到厂商名声的影响，不过你必须有明辨是非的能力，言过其实的传言不在少数。如果听到某个发行套件的传言，请花些时间验证相关信息的真实性。如果这是个商业发行套件，可以跟厂商联络。对方可能知道相关信息的来源，最重要的是，能够对传言提出合理的说明。这种验证过程并不是嵌入式 Linux 发行套件特有的。嵌入式 Linux 发行套件特有的现象是，当商业发行套件的厂商对开放源码社群做出贡献时，便会建立起该商业发行套件的名声，厂商若能够提供更多开放源码软件或资助其开发，表明自己与开放源码社群的关系，就能够站在非常有利的位上，了解这些开放源码计划的变迁与开发将会对其未来的产品及最终的客户造成什么影响。总之，这是了解厂商的关键性环节和依据，可以从此处看出厂商对其提供的软件抱什么态度。开放源码发行套件本身已经符合这个标准，因为其本身就是一个开放源码的贡献。

商业发行套件可能必须提供的另一个重要工具就是文档。在当今这个瞬息万变的年代，能够包含最新信息且内容精确的文档并不多见。开放源码计划若提供了文档，通常都已经过时。Linus Torvalds 的话一点也没错，他说：“用源码，路克”（Use the source, Luke）（译注 9），意思是指倘若需要了解软件，就应该看它的源码。但是要达到精通的程度必须投入相当的时间和精力，因此仍需辅以适当的文档。因为开放源码开发者宁可花时间

译注 9：“用源码，路克”（Use the source, Luke）这句话系源自电影《星球大战》（Star Wars）中，Obi-Wan 对 Luke Skywalker 所说的：“用你的原力，路克”（Use the Force, Luke）。

写程序也不愿意浪费时间写文档，所以“发行套件”厂商必须为其产品提供适当的文档。当你在评估“发行套件”的时候，务必了解其附带文档的类型和完善程度。尽管开放源码发行套件的文档不算多，但与商业发行套件比起来，有些开放源码发行套件提供的文档算是相当优质的了。

若考虑到开发与目标板设置在某些方面的复杂度，开发及（或）目标板发行套件的安装可能不容易。有这方面考虑的话，或许可以选用容易安装的发行套件。尽管这么做并没有错，但不要忘了，一旦发行套件安装好之后，就不应该事后再重新安装。也请记住，正如稍早所说，目标板发行套件实际上并没有所谓的安装，它只是用来让目标板设置容易完成而已，并没有传统上所谓的“安装”程序。在发行套件的安装过程中，应该注意三件事：明确的说明（文字说明是否出现在安装期间、手册中，或二者）、可配置、自动化。可配置能看出你对所安装的套件具备多少控制权，自动化则是指使用内含配置选项的文件让过程自动进行的能力。

由于有些CPU和目标板被广泛地用于嵌入式系统的开发，商业发行套件厂商于是特别针对那些常用的CPU和目标板定制并提供包装好的开发及（或）目标板发行套件。如果想要使用特殊的CPU或目标板，可能需要找到已经为设备做过测试的发行套件。

使用发行套件应避免的事

使用发行套件时应该避免一件事，那就是往后所有的开发计划都只依赖相同的发行套件。不要忘了，使用Linux的主要理由之一，就是你不愿遵从任何人的意愿及市场机制。如果开发计划仅靠发行套件中的私有工具及方法，无疑是让自己陷入“往后所有开发计划都得继续使用相同发行套件”的险境。但这并不是说，不应该使用附带的增值软件（而且其他发行套件中可能找不到这些软件）的商业发行套件。这表示应该有个备选计划，让你使用不同发行套件中的不同工具来达到相同的结果。

以多组件系统为例

为了展示并探讨整本书的内容，这一节将要来检查一个嵌入式Linux系统范例。这个嵌入式系统由多个相互依赖的组件组成，每个组件都是一个单独的嵌入式系统。对用户而言，整个系统具有一组固定的功能，但是每个组件的构成及实现有可能不同。因此，这个范例可以为我们提供许多观点，让我们能够探讨各种解决方案、其损益权衡以及其细节。总之，该系统可以涵盖的嵌入式系统类型最多，从规模非常小到非常大，还包括了用户交互功能和联网功能等多个方面，并且涵盖了时间方面的需求。

一般架构

这个范例用到的嵌入式系统是一个工业的过程控制系统。该系统由多台具有网络功能的计算机组成，这些计算机的操作系统都选用Linux。图1-1展示了此范例系统的一般架构。

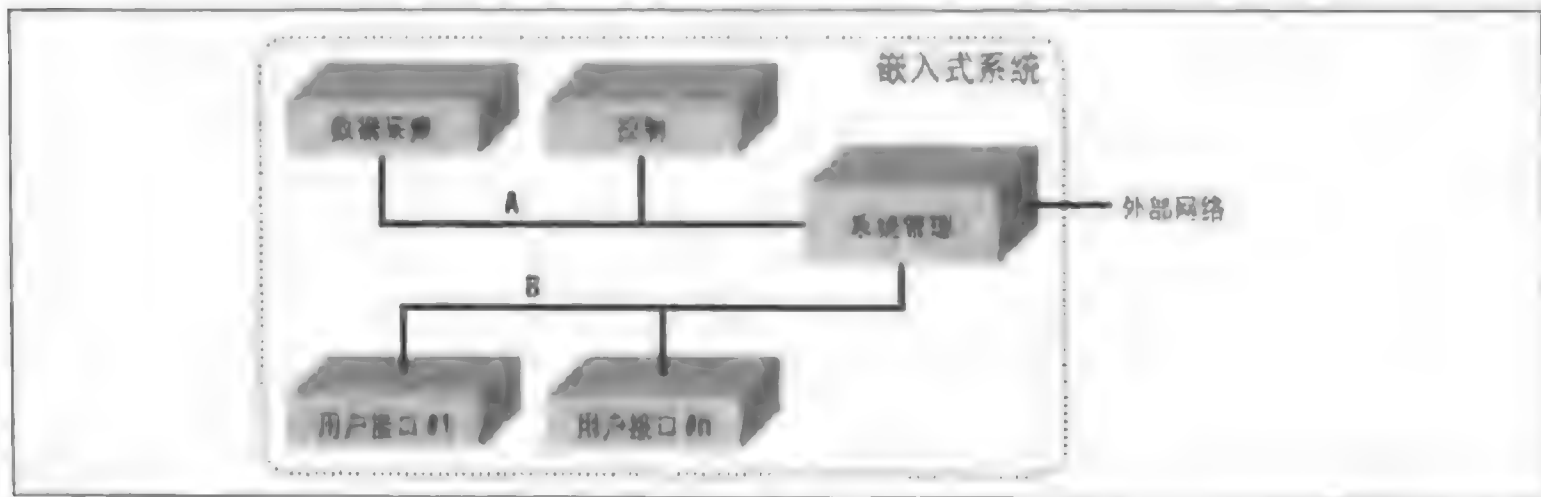


图 1-1: 范例嵌入式 Linux 系统的架构

就内部来说，此系统由四种不同类型的机器组成，每个组件用来满足不同的目的：数据采集（data acquisition, DAQ）、控制、系统管理（system management, SYSM）以及用户界面（user interface, UI）。组件间使用最常见的接口和协议，TCP/IP over Ethernet（Ethernet 上的 TCP/IP 协议），互相联系。在这样的设置中，DAQ 和控制模块专注在 Ethernet 的链接上，UI 模块针对另一条链接。除了作为这两条链接的接口，SYSM 模块通过第三条链接为“外界”（可能是公司内部的网络）提供了一个接口。

该系统控制的程序可能是工厂处理设备的一部分，或是其他完全不一样的东西，不过这与我们要探讨的内容无关，因为所有的过程控制系统都具有类似的架构。为了控制程序，系统时刻都需要知道程序中不同组件的现行状态，这就是 DAQ 模块的功用。获得数据后，系统便能够判断如何控制程序。尽管分析数据的方向可能不同，但所有的控制命令都来自控制模块。因为某些程序的控制常需要人的介入及（或）监控，所以必须让工作人员通过某个渠道参与系统的运行，以便观察及修改程序，这就是 UI 模块的功用。为了将所有组件整合在一起，以及提供集中存储数据的能力和管理接口，SYSM 模块放在所有组件的中心位置上，并且对外界提供进入系统的惟一存取点。

每个组件的需求

每个组件都有一组需求以配合设计上的需要，因此建立的方式也各不相同。以下是每个组件的详细说明。

数据采集模块

过程测量程序的第一个组件是传感器。传感器是一种可以将物理现象转换成电子信号的装置。热电偶、力传感器、加速度计、线性差动变压器（linear variable differential transformer, LVDT）都是传感器，分别可用来量测温度、机械应力、加速度以及位移。传感器通常会直接放在过程发生的区域中。如果有一个用来煮沸液体的熔炉，需要监控其温度，就可以将热电偶放入用来盛放该液体的容器中。

传感器输出的电子信号通常都会经过信号调节的各个阶段，在馈入 DAQ 装置之前，信号可能会经过放大、缩小、过滤及隔离的处理。DAQ 装置，通常是一片安装在计算机中的 DAQ 卡，可用来取样模拟值，并将其转换成数字值，以及将这些值存入取样缓冲区里。然后各种不同的软件组件可利用这些值来画曲线、探测特定条件、或修改特定控制参数以便对信号做出响应，例如在反馈循环中的时候。

市面上已经有一些专门探讨 DAQ 的书，而且本章的目的也不是为了深入探讨 DAQ。更确切的说，我们将会假定所有信号的取样和调节都已经完成。同时，我们不会将讨论局限在特定的 DAQ 卡上，我们将会假定 DAQ 卡的驱动程序符合 Comedi 提供的 API。Comedi 是一个软件套件，可用于数据采集和控制，稍后将会说明。

因此，DAQ 模块应该是一台内含 DAQ 卡，并且通过 Comedi 对程序相关数据进行取样的工业计算机。这台计算机是一个中型的嵌入式系统，对时间有严格的要求，没有用户界面，当要连往系统的其他地方时会使用 Ethernet（注 9）。

典型的配置中，DAQ 模块会将取样数据存入当地的缓冲区。数据分析可以在当地进行，也可以传送到 SYSM 模块进行分析。不管怎样，重要的数据一定会转交 SYSM 模块进行备份，并被各个 UI 模块显示出来。当数据分析在当地进行时，倘若检测到异常或危急的情况，将会通知 SYSM 模块。相对来说，DAQ 模块将会执行 SYSM 模块送来的命令。这些命令包括指定取样率、分析参数，或甚至是一旦分析完之后应该如何处理收集到的数据。由于 SYSM 模块对 DAQ 模块的运行了如指掌，每当有需要或被要求这么做时，DAQ 模块会将运行状态和错误信息转送至 SYSM 模块。

DAQ 模块通常会使用 CompactFlash 或固有的 flash 设备引导，以及使用 RAM 磁盘或 CRAMFS 存储数据。这么做有利于发生硬件故障时更换模块。它的软件配置包括针对 PC 类型的系统或基于 PowerPC 架构的系统建立的抢占式内核。DAQ 并不会对外提供 FTP、HTTP 或 NFS 之类的服务，它执行的是可联系 SYSM 模块让整个系统正常运行的定制监控程序。因为 DAQ 模块并非多用户系统，而且不会有用户直接跟它交互，所以

注 9： 尽管在此范例中并未用到，市面上已经可以看到具 Ethernet 功能的 DAQ 卡。

DAQ 模块只会对用户的工具提供最起码的支持。这可能需要用到 BusyBox 套件。DAQ 使用的是固定的 IP 地址，这是在设计期间决定的。因此，SYSM 模块轻易就能察觉 DAQ 模块是否还处在运行状态。

控制模块

传统的过程控制会用到价格昂贵的可编程逻辑控制器（programmable logic controller, PLC）和类似的系统，执行自己专属的操作系统，以及特殊的配置程序。由于消费市场上出现了价格便宜的新硬件，我们可以更多地看到主流硬件，如 PC，被用在过程控制中。我们甚至可以看到工业级硬件因为使用主流技术的关系而价格滑落。

同样地，过程控制涵盖的领域极广，我也不打算在此处做完整的说明。我会假定此系统控制的硬件可以模拟成状态机。其上的软件会收到与它的控制命令相应的反馈，此反馈是基于状态机现行的状态。

控制模块是一台工业级计算机，它具备可以控制硬件的接口。这台计算机是一个中型的嵌入式系统，对时间有严格的要求，没有用户接口，跟 DAQ 模块非常像，当要连往系统其他地方时会使用 Ethernet。

控制模块的主要工作就是将命令送达所控制的硬件，并监控硬件对命令的反应。通常，这些命令都是 SYSM 模块发出的，SYSM 模块的功能就像系统的决策中心，它会根据从 DAQ 模块获得的数据进行决策。因为 SYSM 模块提供的命令可能包括许多硬件上的操作，所以控制模块将会调节硬件，以便获得 SYSM 要求的结果。一旦操作完成之后，不论发生任何特殊的情况，或是被要求这么做，控制模块均会向 SYSM 模块报告目前硬件操作的状态。

控制模块可以从 CompactFlash 或 CFI flash 设备引导，并可以使用 RAM 磁盘或 CRAMFS，与 DAQ 模块非常像。它使用的平台基于 PowerPC 架构的系统，使用的内核则是具备实时功能的抢占式内核，例如 RTAI 或 RTLinux，因为控制复杂的硬件需要硬性的实时响应时间。所以硬件的控制将会由定制的硬实时驱动程序来完成。同样地，此模块也不会对外提供网络服务。它会通过定制的监控程序与 SYSM 联系，以便让系统的运行协调一致。因为控制模块并非多用户系统，用户也不会直接与其交互，所以将只会提供最起码的用户工具。可能会使用 BusyBox。控制模块也会使用固定的 IP 地址，这么做的理由跟 DAQ 模块一样。

系统管理模块

SYSM 模块的功能是管理及协调系统中各个组件的交互，正如前文所述，它还会为系统提供一个进入点，好让外界使用。此模块是一个大型的嵌入式系统，对时间有严格的要

求，没有用户接口。此模块具备三张网卡：一张用在 DAQ 和控制模块上，一张用在用户接口上，一张用在外部网络上。每个联网接口都具有它自己的一套规则和服务。

在 A 链接上，SYSM 模块会从 DAQ 模块取回数据，存储全部或部分数据，以及将适当的数据转送至各个 UI 模块，以便显示。所存储的数据可以备份起来供未来参阅，或是成为质量控制程序的基础。数据的备份可以使用传统的备份方式来进行，或是使用具有备份程序的数据库。正如前文所说，SYSM 模块可能会对获得的数据进行分析，如果这不是在 DAQ 模块中进行的话。不论分析是否在当地或 DAQ 模块中完成，SYSM 模块都会根据分析的结果和过程控制现行的状态，对控制模块下达命令。SYSM 模块还会执行定制的监控程序与工具程序，以便跟 DAQ 模块及控制模块上的监控程序联系。正如 A 链接上的其他组件一样，SYSM 模块也是使用固定的 IP 地址，这样 DAQ 及控制模块可以轻易就辨认出它来。

SYSM 模块会对外部网络提供 HTTP 及 SSH 服务。HTTP 服务让位于外部网络且经认证的用户通过网页及表单，设定或监控整个系统的各个方面。SSH 服务让嵌入式系统的制造商能够从远程登录系统进行问题排除及升级。在这样的大型系统上若能具备 SSH 服务器，将可以同时降低制造商和客户的维护费用。

SYSM 模块中有一个可设定的选项，就是对外界报告错误的方式。这个选项可用来指示 SYSM，当它无法操作时，如 DAQ 或控制模块故障，要如何处理错误。标准程序可能是在扬声器中发出警报，或者是使用 SNMP 来通知管理员，或是将紧急显示要求送往适当的 UI 模块。对外界的链接是另一个可设定的选项。SYSM 模块可能会使用固定的 IP 地址，或者使用 DHCP 或 BOOTP 来获得其 IP 地址。

在 B 链接上，SYSM 模块会提供 DHCP 服务，让各 UI 模块能够动态配置其地址。UI 模块一旦开启就会向 SYSM 模块注册，SYSM 模块会把数据及系统现在的状态，转送至已注册准备显示的 UI 模块，UI 模块则会根据系统的状态来反应数值的变动。在显示的过程中，工作人员可以根据自己的需要修改所要显示的数据量，SYSM 模块则会开始或停止向 UI 模块转送某数据以响应此设定。

由于 SYSM 模块是一个大型的嵌入式系统，所以它将会使用硬盘来引导，并且具备传统工作站或服务器全部的特性，包括交换的功能。SYSM 模块可能是一部 Sun、PowerPC、ARM 或传统的 PC。SYSM 模块会因为实际使用的平台不同而有些微的差异，这是由于它大部分的功能都相当高级。因为 SYSM 模块需要同时为许多不同的应用提供服务，并且要快速响应进来的流量，所以 SYSM 模块会使用抢占式内核。

用户接口模块

UI 模块让工作人员能够借助检查现行状态反映的值，并且修改控制程序的变量，来与运

行中的程序交互。UI 模块一般会使用小型嵌入式系统，对时间的限制采取宽松的方式。UI 模块也具有网络的功能，但是方法可能各不相同。与前面提到的系统组件相比，UI 模块可以有各种化身。有些可能是固定的，并且就近附接在过程控制中的敏感位置。有些可能是便携式的，让工作人员能够在程序运行时在厂房中带着到处走，并可以键入或取样各种数据。毕竟，过程控制的某些部分或许无法自动进行，可能需要自己手动将数据键入系统。

各 UI 模块显示的数值，是通过联系 SYSM 模块上适当的定制监控程序取回的。当 UI 模块收到紧急事件时会立即显示，UI 装置上执行的定制监控程序会等待 SYSM 模块送来紧急事件。有些信息，如紧急事件，无论如何都会被显示出来，不管是怎样配置的。有些 UI 模块可能只会显示有限的一组数据，有些则会显示正在运行的程序的详细信息。还有些 UI 模块可能处在紧急程序中，正忙于处理紧急状态。

由于 UI 模块规模小，它一般会从 flash 设备或通过网络引导。以后者来说，SYSM 模块必须被设定成能够照顾到远程引导。不论是否使用远程引导，UI 模块都会通过 DHCP 来获得自己的 IP 地址。便携式 UI 模块一般会使用 ARM、MIPS 或 m68k 等架构，并且会执行标准的内核。当 UI 模块与用户交互时处在自动模式，只需要最起码的用户工具，尽管要显示，并且与用户交互都要用到大量的图形工具程序及链接库。因为我们假定只有经过认证的人才可以访问 UI 模块，所以我们并未在 UI 装置上实现任何形式的认证功能，因此所有的 UI 模块都不是多用户系统。不过，可依照系统实际的需求变更做法。

需求变化

以上对各个模块所做的说明，只是用来实现系统的基本方案。不同的组件以及系统的架构都还有变更的空间。以下不按任何顺序列出可能的变化：

- 根据系统部署的实际情况，有可能需要不断验证各个系统组件的连通性。办法就是从其他模块向 SYSM 模块送出 keepalive 信号或是使用看门狗定时器。
- 如果必须在一定的时间范围之内响应一些紧急的情况，那么在 A 链接上使用 Ethernet 上的 TCP/IP 协议可能会造成某些问题。倘若 DAQ 模块观察某个化学反应，得知即将发生灾难、在化学反应失控之前，可能需要通知 SYSM 模块。有这种需要的话，最好的主意就是使用 RTNet，因为 RTNet 会使用 Ethernet 上的 UDP 协议来提供硬实时功能（注 10）。这需要你在 SYSM 模块上运行具有实时功能的内核。

注 10： 尽管 UDP 不像 TCP 那样会延迟包的传送，但是 Linux 中所使用的标准 TCP/IP 栈并不具有硬实时功能。RTNet 所提供的硬实时网络通信，是直接提供 RTAI 或 RTLinux 上的 UDP 堆栈。

- Ethernet并不是适用所有环境。已经知道有些协议在工业环境中使用比较稳定。如果需要的话，设计者可能希望将Ethernet换成已知的工业级联网接口，如RS485、DeviceNet、ARCnet、Modbus、Profibus或Interbus。
- 为了缩小尺寸及速度上的考虑，设计者可能希望将DAQ、控制、SYSM等模块在单一的设备中实现，例如在CompactPCI机架上插入由各个模块制成的插卡。
- 基于管理的目的，将UI模块实现成X终端机可能比较简单。在这种配置中，UI模块的功能只是个显示及输入终端机。所有的计算负荷都落在SYSM模块身上，SYSM模块变成了X应用程序主机。
- 如果系统的规模不是非常大，而且所控制的程序也比较小，那么将DAQ、控制、SYSM等模块全都放进一台足以胜任的计算机，有其实质上的意义。
- 如果一条网络链接不能胜任DAQ模块产生的流量，那么额外加一条专门传送数据的链接，有其实质上的意义。
- 因为基于质量控制的目的保留程序数据的需要会越来越频繁，所以SYSM模块必须使用数据库。这个数据库将会存放与系统中各种操作有关的信息，以及DAQ模块记录的数据。

还可能其他的变动，这取决于系统的需求。

设计与实现方法

嵌入式Linux系统的设计与实现可以依照明确的方法来完成。整个过程包括许多工作，其中有些工作可能会平行完成，因此可以缩短整个开发时间。如果使用发行套件的话，甚至还可以略过其中某些工作。不管使用什么工具或方法，都需要阅读第二章，因为构建嵌入式Linux系统的所有工作都涵盖在这一章中。

在设计及实现嵌入式Linux系统的时候，可以使用附录一提供的工作单来记录系统的特性。附录一还分节描述了嵌入式系统的每个方面。附录一的工作单不仅可以帮助工作团队记录系统的组成组件，也可以协助未来的维护者了解系统最初的建立方式。事实上，一份正确完整的工作单应该足以让工作团队以外的人在没有任何帮助的情况下重建整个系统。

构建嵌入式Linux系统涉及的工作细节，有时会随着相关软件套件的更新而有所变动，请随时访问本书的网站（<http://www.embeddedtux.org/>），以便获得最新的资料。

建立目标板 Linux 系统

欲建立目标板 Linux 系统，必须对各个系统组件进行适当的组合及设定。程序设计及开发工作属于不同的主题，本章稍后会讨论到。

建立目标板 Linux 系统有 4 个重要的步骤：

- 决定系统组件
- 配置及建立内核
- 建立根文件系统
- 设置引导软件与配置

决定系统组件就好像是前往杂货店之前先写下采购单一样。由于 Linux 有许多选项，若不先备好清单，到头来很容易连自己选用的是什么也不知道。这可能会导致一个现象，系统将会拥有许许多多与主要用途无关的功能。因此，当你正准备检查有哪些最新的 Linux 选项可用时，先不要急，坐下来想想你需要什么，并列清单。我发现这么做有助于将焦点摆在开发计划中，避免分散注意力。但这并不表示，即使找到了适当的选项，也不应该变更清单的内容。这只是要你注意到 Linux 的可选用软件非常丰富的事实。

第三章探讨的硬件组件自然也是嵌入式 Linux 系统的一部分。这应该能够提供你足够的背景知识，甚至可以说是让你明白，在嵌入式 Linux 系统中可以找到哪些硬件。由于 Linux 和相关软件将会不断变化，可以据此在网络上做进一步的调查，找出 Linux 符合哪些设计需求。接着，应该能提供一份选项清单，让你能够用来开发整个系统。这个开发步骤只属于单一工作，无法与其他工作一起进行。在你进行其他步骤前，必须先完成“决定系统需求及 Linux 是否符合需求”这项工作。

因为 Linux 是不断发展变化的，你可能会觉得需要为自己的设计获得最新最优秀的软件。请不要这么做，新的软件通常需要测试，并且可能会因为各个软件包之间的依存关系，而导致必须升级其他软件。因此，你可能会发现自己为了跟上大量更新脚步而陷入泥潭。所以倒不如修正你现有软件的缺陷，记下有哪些新增的功能，这样在发展下一代计划的时候，设计利用这些新增的功能。如果基于重要的理由，必须升级某个软件组件，在你实际进行升级之前，请谨慎分析这样的升级会造成何种结果。在应用到主系统之前，或许还应该在“测试系统”上先测试一下升级的结果。

决定好设计应该包含哪些特性之后，接下来可以选择内核的版本与适当的配置。内核的配置与建立程序参见第五章的说明。与其他软件不同的是，从计划的开发到 beta 测试阶段，或许都会想将内核更新到最后的稳定版本。尽管要在整个开发周期中维持内核版本的稳定似乎很简单，但你可能会发现，必须依照内核最近几个版本所做的修正，自己修

正内核的缺陷。正如第五章所说，必须随时了解内核发展的最新状态，以便有办法判断更新到最新的内核是否对你最好。同样地，你或许也会想要试用一下较新版的内核，如果遇到任何严重问题，还可以回头使用较旧版的内核。切记，使用太旧版的内核可能无法得到社群的支持，因为贡献者们恐怕不会响应与旧缺陷有关的问题。

无论你是否要跟上内核更新脚步，都建议最好在整個计划中维持内核配置的一致性。这将可以避免在开发期间破坏完整性。然而这涉及到根据系统需求详细调查配置选项。尽管此工作可以跟其他工作同时进行，不过重点在于参与计划的开发者们能够意识到有哪些可能的配置选项，以及对选项的决定有一致的看法。

一旦决定了配置之后，接着就是建立内核。建立内核包括许多步骤，所产生的也不只是内核映像而已。尽管产生的组件在计划的某些开发阶段用不到，但是当计划进一步发展时，你会发现计划中其他组件会越来越依赖内核组件的可用性。因此最好能充分地设定好内核组件，而且越早建立越好，并且要在整个计划中维持其最新的状态。

在处理内核问题的同时，可以按照第六章的说明，开始建立嵌入式系统的根文件系统。嵌入式Linux系统的根文件系统与运行Linux的工作站或服务器类似，不过嵌入式Linux系统只包含系统运行必需的应用程序、链接库和相关文件的最小集合。注意，在这个阶段你不要为了获得大小适当的根文件系统，而移除之前选用的任何组件。事实上，如果真的这么做，也许意味着你选用的系统组件并不适当。不要忘了，之前的阶段应该包含对所有系统需求的分析，其中包括根文件系统的大小。因此，在建立目标板系统时，每个组件的大小估计值应该越精确越好。

如果无法为嵌入式系统中将会用到的组件预先决定完整的清单，建立目标根文件系统的时候，倒不如以反复的方式，陆续加入你所需要的工具程序和链接库，不过千万不要将它视为最后的根文件系统。你反而应该用这种反复的方式探索根文件系统建立的过程，然后用这个经验为目标板系统建立完整的根文件系统。其背后的理由是，反复的本质是不断摸索，这使得它的完成时间遥遥无期。这种建造根文件系统的方法可能需要更多的事先考虑，但是它的结果是可预期的，也可以作为额外计划的依据。

在建立目标板Linux系统的过程中，还剩下设置及配置存储设备与引导加载程序等工作。第七、八和九章的全部内容都在探讨这些问题。在这些步骤中，目标板系统包含的各个组件，引导加载程序、根文件系统、内核，将会被组合起来。因平台的不同，引导时将会用到不同的引导加载程序。就单一架构来说，不同的引导加载程序在调试和监控的能力上也会有所差异。不同的架构之间，系统的包装和引导方法相当类似，主要的差别在于引导时使用的永久性存储设备以及引导加载程序。例如，从原生的flash设备引导就和从DiskOnChip或CompactFlash设备引导不同，而且与从网络服务器引导差别更大。

开发工具的设置与使用

嵌入式系统的软件开发与工作站或服务器环境上的软件开发不同。重点在于，目标板的开发方式通常跟在主机上不同。因此，需要对主机/目标板进行设置，让开发者能够在主机上开发自己的软件，然后将软件下载到目标板上进行测试。这种设置有两个阶段：开发与调试。然而，这样的设置并不会阻碍你利用 Linux 多架构的优点，在不需要修改或只须一点修改的情况下，在主机上测试目标板的应用程序。尽管并不是所有的应用程序都可以用这种方法进行测试，不过在主机上测试目标应用程序通常将可为你节省许多宝贵的时间。

关于嵌入式开发，请参考第四章的说明。在嵌入式系统上测试任何程序代码之前，必须先建立好主机/目标板的联机。开发者将会通过一条缆线与目标板系统交互，并且验证应用程序中开发的功能是否符合规定。应用程序通常无法直接在硬件上执行，目标板上必须已经存在可用的嵌入式 Linux 系统。因为通常不可能等到最终的目标板设置完成之后才测试目标板应用程序，所以你可以利用开发时的目标板设置。后者的包装将会松散许多，不需要顾虑最终套件所需遵守的大小限制。因此，开发时的根文件系统与最终的根文件系统相比，可能会包括更多的应用程序和链接库。这也就是为什么会容许开发期间使用不同且较大的永久性存储设备。

欲编译目标板应用程序和链接库，需要这样的设置：为交叉开发设定或建立各种的编译器及二进制工具程序。有了这些工具程序，可以为目标板建立应用程序，因此开发时的目标板设置可用于往后的开发工作上。完成设置后，可以使用各种集成开发环境（Integrated Development Environment、IDE）来简化计划组件的开发工作，并使用其他工具，例如使用 CVS 跟别的开发者进行协同开发。

考虑到嵌入式系统本身具备的能力，有些开发者甚至选择直接在目标板系统进行所有的开发工作。在这样的设置中，编译器和相关工具程序全都在目标板上执行。这其实就是将主机与目标板结合在一部机器中，这类似于传统的工作站应用程序开发。这种配置的主要优点在于，可以避免设置主机/目标板环境的麻烦。

不论开发时是如何设置的，都需要对软件进行调试与修正。方法很多，可以使用第十一章提到的调试工具。欲进行简单的调试，可以选用特别的方法，例如使用 `printf()` 打印出值来。有些问题需要深入观察软件执行时的运行情况；可进行符号调试。对 Linux 来说，gdb 算是最常见的通用调试器，不过在嵌入式系统上进行符号调试可能比较麻烦，它可能涉及到远程串行调试、内核调试，以及 BDM 和 JTAG 等调试工具。但即使是符号调试，有时也可能不管用。当应用程序进行的系统调用有问题时，或当有同步的问题需要解决时，最好使用 *strace* 和 LTT 之类的跟踪工具。至于性能方面的问题，还有其他

更适合的工具，例如 *gprof* 和 *gcov*。除了以上这些问题，甚至可能需要了解内核崩溃的原因。

为嵌入式系统开发软件

使用 Linux 作为嵌入式操作系统的主要好处之一，就是为 Linux 开发的程序代码，不管是在工作stations上或是在嵌入式目标板上，运行情况应该都一样，对吧？没错，但不完全是这样。虽然这样没错，Linux 工作stations上建立的程序代码，同样可以在嵌入式 Linux 系统上运行，但是嵌入式系统的操作及需求跟工作stations或服务器环境有很大的不同。例如，在工作stations上，可以在预期的错误发生时，终止应用程序的执行状态，并将重新启动应用程序的责任交给用户，不过在嵌入式系统中恐怕没办法这么做，但你也不容许应用程序无止尽地攫取系统资源或行为失当（注 11）。因此，即使它们所使用的 API 和操作系统可能一样，但它们设计程序的哲学基本上是不同的。

网络功能

网络功能让嵌入式系统能够跟外界交互。在嵌入式 Linux 环境中，当有网络安全考虑时，必须慎重选择网络硬件、网络协议以及所提供的服务。第十章将提到网络服务（如 HTTP、Telnet、SSH 及（或）SNMP）的设置与使用。在具有网络功能的嵌入式系统中，值得注意一件事，那就是远程更新的可能性，有了这个功能，就可以通过网络更新系统，不必亲自到现场来做。详情参见第八章。

注 11：Linux 工作stations和服务服务器上，一般的应用程序也不应该无止尽地攫取系统资源。事实上，最重要的是，Linux 服务器上所使用的应用程序以稳定著称，这也就是 Linux 作为服务器操作系统如此成功的一个理由。

第二章

基本概念



正如前一章所见，嵌入式Linux系统的种类繁多。不过我们却可以找到几个大多数嵌入式Linux系统适用的关键特性。本章的目的在于对读者说明这些基本概念，以及当读者在开发任何一种嵌入式Linux系统时可能遇到的问题。

本章介绍的主题中，有许多将会在后面其他章节作更深入的探讨。之所以要先在本章介绍这些内容，主要是让读者能够通盘了解整个系统的架构。

本章一开始会分节探讨开发嵌入式Linux系统最常用的主机类型，主机/目标板开发环境的设置类型，以及主机/目标板调试环境的设置类型。以上这几节的目的在于帮助读者选择用来开发嵌入式Linux系统的最佳环境，或者如果开发环境已经确定下来并且无法改变的话，读者也可以藉此了解自己特有的设置，往后将会对开发结果造成何种影响。接着本章会详细介绍嵌入式Linux系统中最常见的结构。我在这个部分会介绍嵌入式Linux系统的一般架构，并说明系统的启动方式、引导配置的类型，以及典型的系统存储地址布局。

主机类型

第三章涵盖了嵌入式Linux目标板中最常见的硬件。这些目标板系统可以通过各式各样的主机来进行开发。接下来，我将会探讨最常用的主机类型、这些主机的特点，以及通过这些主机来开发嵌入式Linux系统是多么容易。

Linux 工作站

用来开发嵌入式Linux系统的主机，以Linux工作站最为常见，它也是我推荐的主机类型，这是因为要开发嵌入式Linux系统，就必须对Linux相当熟悉，而且熟悉Linux的最好方式，就是把它作为你每天工作的平台。

你使用的Linux工作站通常就是一台标准的PC。但是不要忘了，Linux可以在各式各样的硬件上执行，所以你不一定要使用PC。例如，习惯上，我会使用Apple的PowerBook来为自己的嵌入式开发工作运行Linux。虽然PowerBook上没有RS232，不过可以使用USB串行转换器来解决此问题。

你可以在自己的主机上使用任何标准的Linux发行套件，如Debian、Mandrake、Red Hat、SuSE或Yellow Dog。事实上，整本书都会假定你执行的是一般的发行套件。正如第一章所说，我不会限定你使用哪种嵌入式Linux发行套件来开发嵌入式Linux系统。本书只会提供所有必要的信息，让你能够建立自己的开发环境。

注意： 尽管我极力在书中保持主机与发行套件的无关性，但是书里所用到的指令还是有一点偏向Red Hat类型的发行套件。所以在书中出现的命令有些可能需要略微地修改才能正常执行，这取决于你安装的是哪种发行套件。请注意，随处都有可能出现与发行套件相关的命令。

当然，最新最快的硬件是每个工程师梦寐以求的。尽管拥有最快的机器必然会对工作有所帮助，但是对这类开发工作而言，其实只要RAM搭配得当，使用慢一点的机器也无所谓。不要忘了，Linux具有善用硬件的能力，例如，我常会使用Pentium II 350MHz的系统搭配上128 MB的RAM来进行开发工作。

在磁盘与RAM方面，将会需要大量的存储空间。除了发行套件使用的空间，开发环境及项目的工作空间即使需求不多，也应该设置2或3GB的磁盘空间。例如，一个未压缩的内核源码树，将会是包含在项目工作空间中的众多组件之一，编译之前这个部分会使用100 MB以上的空间。编译之后，将会进一步增大。如果一次同时对三到四种内核版本进行实验，光是内核这部分就需要500 MB以上的磁盘空间。

至于RAM，GNU工具链的某些编译步骤需要用到大量的内存空间，尤其是在C链接库建立期间。我建议主机上要使用128MB的RAM以及128MB的交换空间。

Unix 工作站

根据开发环境，可能需要使用传统的Unix工作站。比方说，电信解决方案开发者们就常用Solaris工作站。尽管开发嵌入式Linux系统最常用的还是Linux工作站，但仍有可能使用传统的Unix工作站。

因为Linux本身与Unix非常相似，所以对Linux适用的通常对Unix也适用。GNU开发工具链尤其如此，因为主要的GNU工具如编译器、C链接库、二进制工具程序（所谓的binutils）甚至在Linux出现之前就已经在传统的Unix系统上开发并使用了。

因此，本书接下来要探讨的内容也应该适用于任何 Unix 工作站上。我说“应该”是因为，可能你自己的解决方案略微不同。前面我对 Linux 工作站关于存储空间的建议同样适用于 Unix 工作站。

Windows（2000、NT 和 98 等）工作站

嵌入式系统的开发差不多在 10 年以前转向 Windows 工作站。许多开发者因而变得习惯在这种平台上工作，许多新的开发者甚至一开始就在这种平台上进行嵌入式系统的开发。具有讽刺意味的是，基于这些或其他理由，有些开发者会想要继续使用 Windows 工作站来开发嵌入式 Linux 系统。

这个平台的主要问题似乎是没有 GNU 开发工具链可用。这一点也不成问题，因为 Red Hat 提供的 Cygwin 环境，就是适合在 Windows 上使用的 GNU 工具链，已经有人用它来为 Linux 建立跨平台的工具。你可以在 <http://www.nanotech.wisc.edu/~khan/software/gnu-win32/cygwin-to-linux-cross-howto.txt> 上找到 Munit Khan 提供的“如何在 Windows 主机上为 i386 Linux 目标板建立跨平台开发工具链”的详细过程。尽管在其他 Linux 目标板应用这些步骤的结果如何还没有正式报告，不过也看不出有什么不可行之处。

如果真的需要继续使用 Windows 工作站，而且想很容易就能在 Linux 环境中开发 Linux 目标板，你可能会想要使用模拟及虚拟软件，比如 VMWare 或 Connectix。这样，尽管主要工作站仍是 Windows，不过你却可以在虚拟环境中执行 Linux。

但是，不要忘了，继续使用 Windows 作为每天工作的平台，将无法了解错综复杂的 Linux。因此，你可能会觉得难以理解 Linux 目标板上发生的某些问题。同时，可能需要提升工作站的处理能力和存储空间，以便建立足够使用的工作环境。

主机 / 目标板开发设置的类型

开发嵌入式 Linux 系统有三种不同的主机 / 目标板架构：连接式设置、可抽换存储装置设置、独立式设置。实际设置有可能包括多种架构，或甚至会随时间改变，这取决于需求和开发方法。

连接式设置

在这种设置中，目标板和主机会一直被缆线连接在一起。此连接通常就是一条串行线或是一条 Ethernet 连接。这种设置的主要特色是，目标板与主机之间并未用到实际的硬件存储装置来传送数据。所有数据都是通过连接传送的。参见图 2-1。

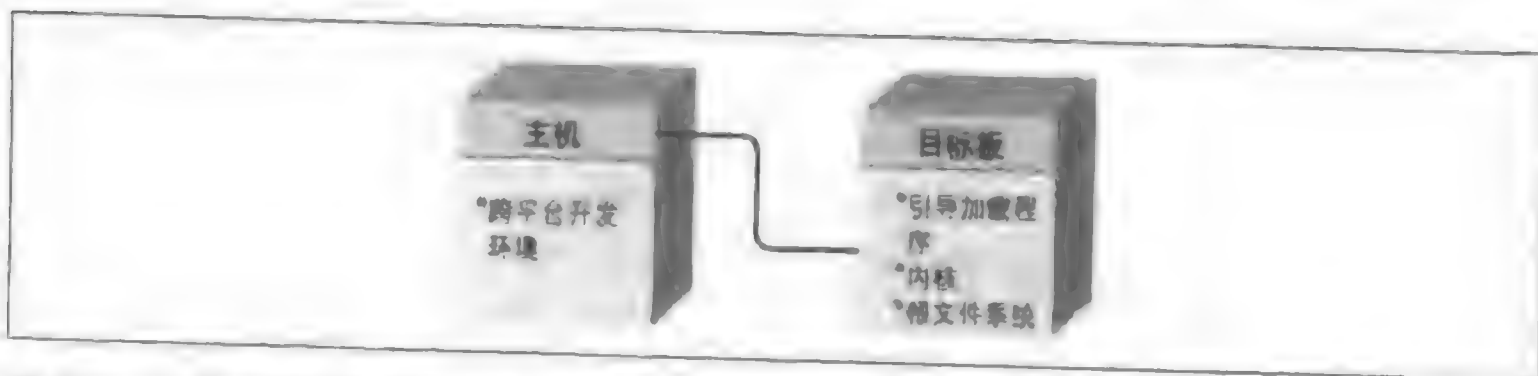


图 2-1: 主机/目标板采用连接式设置

如图 2-1 所示，主机包含了跨平台开发环境（我们将会在第四章探讨相关议题），而目标板则包含了适当的引导加载程序、可用的内核，以及最起码的根文件系统。

另一种做法是，以远程组件来简化目标板的开发工作。例如，可以通过 TFTP 下载内核，此外，根文件系统还可以通过 NFS 安装，而不必在目标板中使用存储媒体。在开发期间使用经 NFS 安装的根文件系统确实相当理想，因为这样可以免除必须不断在主机与目标板之间复制程序变动的麻烦，我们将在“引导配置的类型”一节中描述。

连接式设置是最常见的架构。显然，连接还可以用来进行调试。然而，较常见的设置是使用另一条连接进行调试，我们将在“主机/目标板调试设置的类型”一节中描述。例如，许多嵌入式系统都会同时提供连接 Ethernet 和 RS232 的能力。在这样的设置中，Ethernet 连接会被用来下载可执行文件、内核、根文件系统，以及其他从主机与目标板间快速数据传送受益的大型条目，而 RS232 连接则用于调试。

可抽换存储装置设置

在这种设置中，主机和目标板之间没有实际的连接，取而代之的是，先由主机将数据写入存储装置，然后将存储装置转接到目标板，并用该存储装置引导。参见图 2-2。

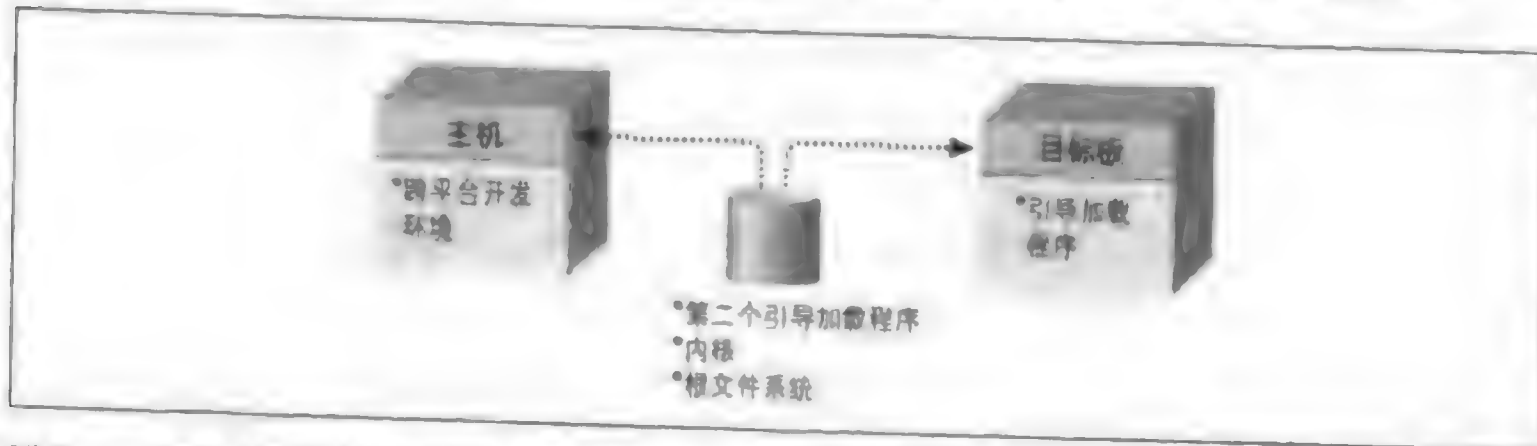


图 2-2: 主机/目标板采用可抽换存储装置设置

与前一种设置相同，主机包含了跨平台开发环境，而目标板则只包含了最起码的引导加载程序。其余的组件被存放在抽换式存储媒体上，例如 CompactFlash IDE 装置或其他

类型的磁盘。先由主机将这些组件写入抽换式存储媒体，然后由目标板上最起码的引导加载程序在启动时加载。

事实上，目标板有可能不包含任何形式的永久性存储装置。例如，目标板有可能使用容易插拔Flash芯片的插座来取代固定的Flash芯片。通常的操作方式为，先在主机上用Flash编程器将数据写入芯片，然后再将该芯片插入目标板上的插座中。

这是嵌入式系统开发初期最受欢迎的一种设置。一旦最初的开发阶段结束，可能会发现换成连接式设置比较实用，因为这样可以避免在每次内核或根文件系统有变动的时候，就需要插拔存储装置来传送数据。

独立式设置

在这种设置中，目标板是个独立的开发系统，它包含了引导、操作以及开发额外软件所必需的任何软件。本质上，这种设置就像在使用一台工作站，只不过其底下的硬件并非传统工作站，而是嵌入式系统本身。参见图2-3。

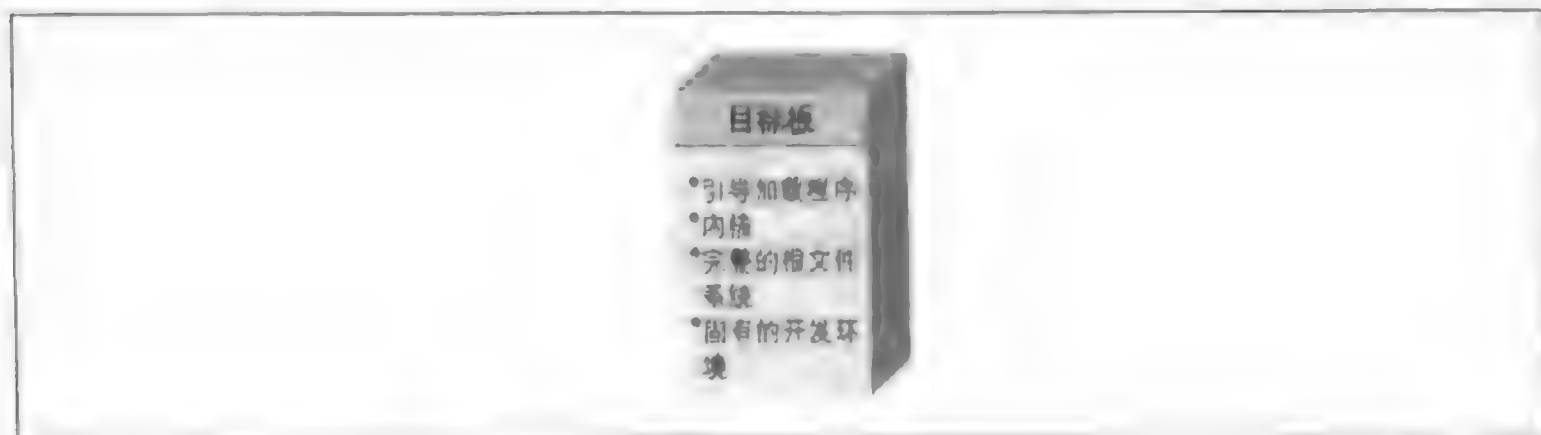


图2-3：主机/目标板采用独立式设置

与其他设置不同的是，这种设置不需要任何跨平台开发环境，因为所有的开发工具都会在固有的环境中执行。而且，目标板与主机之间不需要任何数据传送工作，因为所有必要的存储装置都放在目标板当地。

这种设置非常适合在以PC为主的高级嵌入式系统（例如，高可用性系统）开发中应用，因为开发者可以在嵌入式系统上使用现成的一般Linux发行套件。一旦开发完成，开发者可以根据自己的用途对发行套件进行修整和定制。尽管开发者能够因此避免自己建立根文件系统及设定启动程序，但开发者仍需对自己所使用的发行套件了如指掌。如果对此方法感到兴趣，请参考《Running Linux》这本书。如果准备使用Red Hat，请参考《Learning Red Hat Linux》这本书。这两本书都是由O'Reilly出版的。

主机 / 目标板调试设置的类型

开发者用来连接目标板与主机进行调试的接口基本上有三种类型：串行线、网络接口、特殊的调试硬件。每种调试接口各有其优点及适用范围。我们将会在第十一章探讨部分接口的使用细节。而本节则会简述每种接口的优点和特性。

从主机对目标板进行调试时，最简单的做法就是使用一条串行连接，因为串行电路很简单，而且它通常会以某种形式出现在嵌入式系统中。然而，使用串行连接会遇到两个潜在的问题。其一，串行连接的速度受到限制。其二，如果嵌入式系统中只有一个串行端口，或者串行连接是嵌入式系统对外惟一的接口，那么就不可能在系统调试的同时，以终端仿真器跟系统交互。然而，不具备终端交互能力有时并不是问题。例如，当你用远程内核调试器对内核的启动程序进行调试时，并不需要用到终端仿真器，因为在内核完成引导之前，目标板是不会执行 shell 的。

使用网络接口，例如 Ethernet 上的 TCP/IP 协议，与串行连接相比，可以提供高得多的带宽。此外，目标板与主机可以在相同的物理网络连接上使用多重网络联机。因此，当你对目标板上的应用进行调试时，还可以继续跟目标板交互。当你用终端仿真器通过嵌入式系统的串行端口与目标板交互时，还可以通过网络连接进行调试。然而，有网络接口可用就代表存在协议堆栈。因为协议堆栈就放在 Linux 内核里，所以网络连接无法用来对 Linux 内核进行调试。相对而言，内核的调试通常可以通过串行连接来进行。

使用串行连接和网络接口，需要最起码的软件来处理目标板上最原始的 I/O 硬件。在某些情况下，例如将 Linux 移植到新的目标上或是对内核本身进行调试，就不是这样了。在那些情况下，需要使用可以直接用硬件控制软件的调试接口。有几个方法可以完成此事，不过价格多半很昂贵。

目前，如果要直接控制硬件来进行调试，通常会使用 BDM 或 JTAG 接口。这些接口依靠的是 CPU 硅芯片内嵌的 BDM 或 JTAG 特殊功能。只要将一个特殊的调试器连接到 CPU 上的 JTAG 或 BDM 相关管脚，就可以完全控制 CPU 的行为。因此，当遇到新的嵌入式目标板，或是对目标板上的 Linux 内核进行调试时，通常会使用 JTAG 和 BDM。

就技术原理来说，尽管 BDM 和 JTAG 调试器较内部电路仿真器（In-Circuit Emulator, ICE）便宜很多也简单很多，不过仍然需要购买特殊的硬件和软件（注 1）。通常，这类软件和硬件还是比较贵的，因为 CPU 制造商并不想与别人分享如何使用其产品内嵌的

注 1：如果对嵌入式系统进行调试的各种常用硬件工具（包括 ICE）并不熟悉，可参考附录二所列的相关书籍。

JTAG和BDM接口的详细信息。要获得这些信息，通常会涉及你与制造商的信赖关系以及必须签署严格的保密条款。

让工程部门每个人都配备一个BDM或JTAG调试器的代价或许太大了，不过我强烈建议整个项目至少要配备一个这种调试器，可用来对串行或网络调试器无法处理的棘手问题进行调试。然而，当选用这种调试器时，可能需要评估与GNU开发工具链的兼容性。例如，某些BDM和JTAG调试器，需要使用经过特别修改的gdb调试器。一个优秀的BDM或JTAG调试器应该能够完全处理标准的GNU开发工具链及其产生的二进制代码。

嵌入式Linux系统的一般架构

既然Linux是由许多组件组成的，让我们来对一般Linux系统的架构做一番全面检查。这让我们能够摸清每个组件的定位，了解组件彼此间的关系以及将它们妥善组合在一起的方法。图2-4展示了一般Linux系统的架构及其包含的每个组件。尽管此图高度抽象地表示了内核的内容及其他组件，但是足以用来进行以下的讨论。请注意，与工作站或服务器系统相比，接下来关于嵌入式系统的说明将会有些微小的差异，因为任何Linux系统被高度抽象表示后，看起来都一样。然而，本书其他地方将会强调此架构在嵌入式系统中的应用细节。

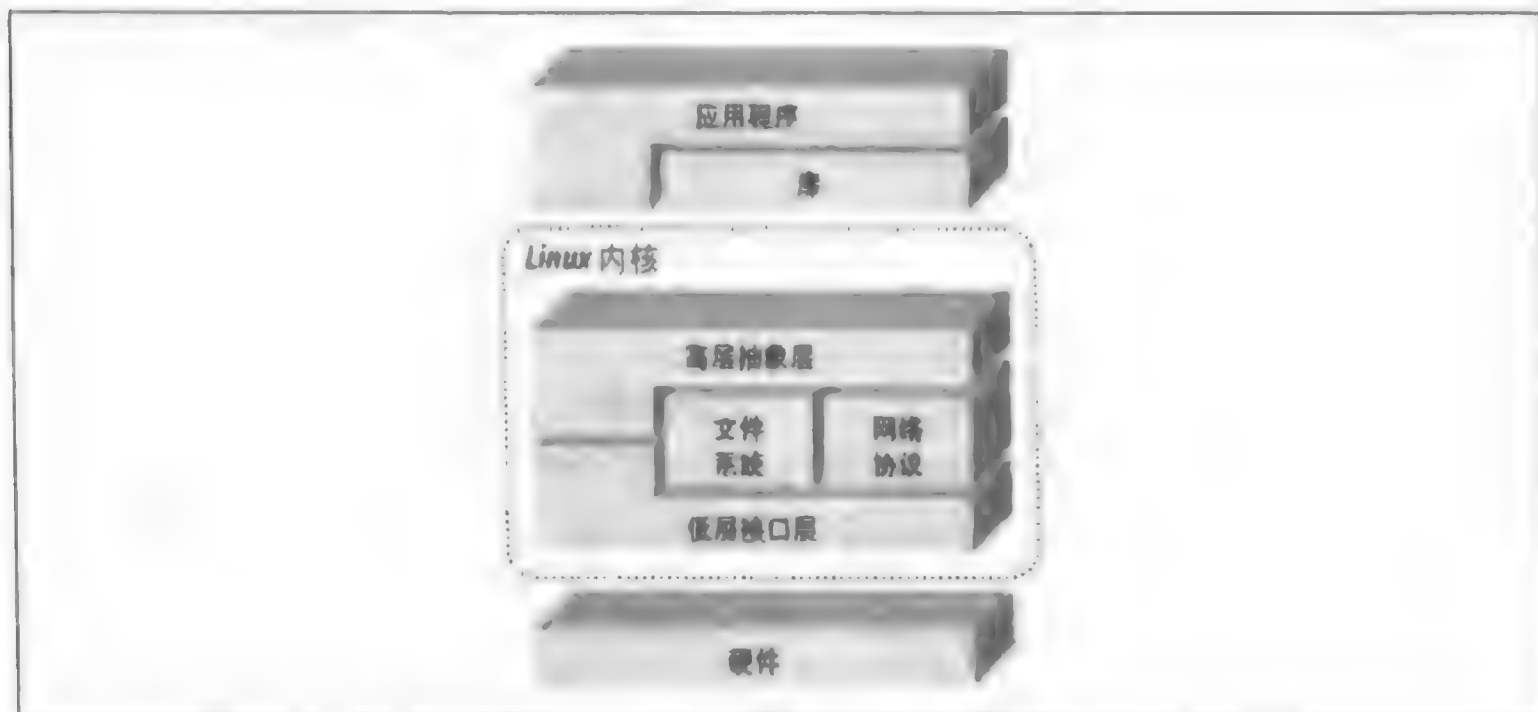


图 2-4：一般 Linux 系统的架构

目标板的硬件必须符合一些要求方能执行 Linux 系统。第一，Linux 需要至少 32 位的

CPU，而且CPU必须配备存储管理单元（memory management unit，MMU，注2）。第二，RAM的容量必须满足系统的需要。第三，如果任何开发工作都是在目标板上完成，而且目标板具备适当的调试功能，则需要最起码的I/O能力。如果必须实地进行问题排除，这一点也相当重要。最后，内核必须能够通过某些形式的永久性 or 网络存储装置来加载及（或）存取根文件系统。参见“嵌入式Linux系统的类型”中探讨的典型系统配置。

硬件的上面就是内核。内核是操作系统的中心组件。使用内核的目的是希望以一致的方式管理硬件，以及为用户软件提供高层抽象层。正如其他的类Unix内核那样，Linux会驱动设备、管理I/O的存取、调度进程、共享存储空间、管理信号的配送以及处理其他管理工作。如果应用程序使用的是内核所提供的API，则应用程序可能根本不用或只需要一点修改就可以移植到此内核所支持的任何架构上。这正是Linux的优势所在，Linux支持的所有架构上可以看到大量相同的应用程序。

为了给应用程序提供它们需要的功能，内核被大致分成两个部分：底层接口层、高层抽象层。底层接口层专属于硬件配置，内核运行在底层接口层之上，并以硬件无关的API（高层抽象层）提供对硬件资源的直接控制。也就是说，对寄存器或内存分页的处理，在PowerPC系统和ARM系统上，会以不同的方式完成，但是却可以使用通用的API来存取内核里高层的组件，尽管有些罕见的例外。通常，底层部分将会处理CPU特有的操作（运算）、架构特有的内存操作，以及设备的基本接口。

在内核提供的底层服务之上，较高层的组件可以对所有Unix系统提供通用的抽象概念，包括文件、socket及信号。由于内核提供的底层API不因架构的不同而不同，所以较高层抽象概念的程序代码实现基本上与底下的架构无关。不过，正如一开始所说，仍会有一些罕见的例外，因为较高层的内核程序代码将会为某些架构包含特殊或不同的功能。

在底层接口层与高层抽象层之间，内核有时需要用到称为“解释组件”的功能来了解如何与特定设备的结构化数据交互。文件系统类型和网络协议就是这类结构化数据的主要来源，内核必须知道如何与这些结构化数据交互。

磁盘一直都是计算机化数据的主要存储媒体。然而，磁盘装置及同样性质的所有其他存储装置中都包含了一些结构。你可以通过指定正确的磁盘、柱面、扇区找到特定的内容，不过这种层次的结构根本无法满足文件及目录变化的需要。要做到文件层次的存取，必

注2：稍后我们将会看到一个经过特别修改的Linux版本（称为uClinux），它可以在一些未配备MMU的CPU上执行。然而，在此类处理器上Linux应用程序的开发与标准Linux应用程序的开发截然不同，需要分开讨论。因此，我将不会探讨如何在没有MMU的架构上使用Linux。

须在磁盘上使用特殊的数据结构，并以特殊的形式存入文件和目录信息，以便日后读取时能够分辨。这就是文件系统的作用所在。然而，由于操作系统发展太快，出现了各种不兼容的文件系统。为满足既有及新开发的文件系统的需要，内核具备了一些可用来分辨特殊磁盘结构的文件系统引擎，并通过此结构取回或加入文件和目录。这些引擎都会对内核的较上层提供相同的API，这样，就算根据文件系统的结构采用不同的底层服务，存取各种文件系统的方法也都是是一致的。例如，内核的虚拟文件层提供的API，对FAT文件系统和ext2文件系统来说都一样，但是块设备驱动程序的磁盘操作会因为FAT和ext2用来将数据存入磁盘的结构而有所不同。

在正常操作期间，内核至少需要一个具有合适结构的文件系统——根文件系统。内核会从这个文件系统加载它要在系统上执行的第一个应用程序。内核还得靠这个文件系统做更进一步的操作，如加载模块以及为每个进程提供工作目录。根文件系统不是存储并且运行在实际的硬件存储装置上，就是在系统启动期间被加载并且运行在RAM上。正如我们稍后所见，在JFFS2文件系统之类的功能出现之后，前者变得越来越受欢迎。

正如所料，内核上面是由操作系统执行的应用程序和工具程序组成的。可是内核提供的服务通常不适合让应用程序直接使用。应用程序应该靠链接库提供的普通API以及可代替应用程序跟内核交互来获得所需功能的抽象服务。大多数Linux应用程序使用的主要链接库就是GNU C链接库。正如我们稍后所见，为了弥补GNU C链接库的主要不足（它的大小），嵌入式Linux系统会将它替换成符合需要的链接库。除C链接库之外，还有其他各种用途的链接库，例如Qt、XML或MD5。

链接库一般会跟应用程序动态链接在一起。也就是说，链接库并不是应用程序的二进制文件的一部分，它们会在应用程序启动期间被加载到应用程序的内存空间。这让多个应用程序能够使用同一个链接库实体，而不是每个应用程序都必须拥有自己的副本。例如，C链接库只会从系统的文件系统加载系统的RAM一次，所有使用该链接库的应用程序都会共享同一个副本。但请记住，在嵌入式系统中，某些情况下会希望链接库成为应用程序的二进制文件的一部分，此时就会使用静态链接而不使用动态链接。例如，当只有链接库的某部分被一或二个应用程序使用时，静态链接可避免必须将整个程序库存放在嵌入式系统的存储装置上的情况。

系统启动过程

在系统启动过程里，有三个主要软件组件参与其中：引导加载程序、内核、init进程。引导加载程序是系统启动过程中执行的第一个软件，它与目标板的硬件有高度的依存关系。正如我们将在第九章所见，Linux有许多引导加载程序可用。引导加载程序在完成底层硬件初始化工作后会接着跳到内核的启动程序代码执行。

内核一开始的启动程序代码会因架构不同而有很大的差异，而且在为C程序代码设置适合的执行环境之前，它会先为自己进行初始化工作。完成以上工作后，内核会跳到与架构无关的`start_kernel()`函数执行，此函数会初始化高层内核功能、安装根文件系统，以及启动`init`进程。

我不准备详细说明内核内部的启动和初始化过程，因为《Linux Device Drivers》（译注1）（O'Reilly）的第十六章都已经说得很清楚了。同时，《Understanding the Linux Kernel》（译注2）（O'Reilly）的附录一，对PC系统从引导到`init`进程执行的整个启动过程，也有很详尽的交代。它探讨的是x86架构上内核内部的启动过程。

系统启动过程的其他部分是由根文件系统上的`init`程序在用户空间中处理的。我们将会在第六章探讨`init`进程的设置与配置。

引导配置的类型

系统的引导配置与所选用的引导加载程序、它的配置以及主机中软硬件的类型有非常密切的关系。例如，网络引导配置需要主机给目标板提供某些类型的网络服务。设计系统的时候，首先要找出可能会在开发期间及最后的产品中使用的引导配置。然后，必须选出一个或一组引导加载程序，以迎合你可能会用到的各种引导设置。例如，并非所有的引导加载程序都可以从磁盘装置引导内核。接下来，我将会说明各种可能的引导配置。不过，在此之前，我会先介绍一些与引导有关的基础知识。

任何CPU都会从制造商预先指定的一个地址获得第一道指令。任何用CPU构建成的系统都会在该位置上使用某种形式的固态存储装置。此存储装置传统上会使用屏蔽式ROM，但现在有越来越多使用flash芯片的趋势（注3）。系统的引导工作将会交由此存储装置上的软件负责。引导软件的复杂度以及引导之后参与系统操作的程度与系统的类型有相当大的关系。

在大多数工作站和服务器的上，引导软件只负责从磁盘加载操作系统，以及给操作员提供基本的硬件配置。相对而言，在嵌入式系统中，如果应用不同，则引导软件的用途很少会相同。有时引导软件在系统运行期间都会一直运行。引导软件还可能是一个简单的监

译注1： 该书中文版《Linux设备驱动程序》已由中国电力出版社出版。

译注2： 该书中文版《深入理解Linux内核》已由中国电力出版社出版。

注3： 当装置的产量非常大时还是会继续使用屏蔽式ROM。例如，消费性游戏产品，如游戏机，通常都会使用屏蔽式ROM。

控程序，负责加载其余的系统软件。这类监控程序可以提供增强的调试和升级能力。引导软件甚至可能会加载额外的引导加载程序，这种情况通常发生在x86PC上（译注3）。

嵌入式Linux系统与它们的非Linux同类系统并不相同。嵌入式Linux系统的特点是在需要时加载Linux内核以及指定的根文件系统。正如我们将看到的，内核的加载及根文件系统的操作受到系统需求（有时还包括系统的开发状态）的影响很大，参见“主机/目标板开发设置的类型”一节的说明。

嵌入式Linux系统的引导可以有三种设置方式：固态存储媒体，磁盘以及网络。每种设置都会有自己特有的配置和用法。接下来将会分节探讨这些设置的细节。

我们将会在第九章为下面所描述的每种设置探讨特定引导加载程序的设置与配置。

固态存储媒体

在这种设置中，固态存储装置用来存放最初的引导加载程序，它的配置参数、内核以及根文件系统。尽管嵌入式Linux系统在不同的开发阶段可能会使用其他的引导设置，不过系统完成开发后多半会使用固态存储媒体来存放所有的系统组件。从图2-5中可以看到固态存储装置对所有系统组件典型的安排方式。



图2-5：固态存储设备媒体对所有系统组件典型的安排方式

图2-5并未显示内存地址，因为寻址范围的差异实在太大了。直觉上，你可能会认为左手边是较低的地址，而右手边是较高的地址。然而，有的时候却刚好相反，而且引导加载程序位于存储装置顶端，而不是底部。因此，许多flash装置会同时提供顶端引导（top-boot）和底部引导（bottom-boot）的配置。flash装置上引导加载程序所在之处（依配置而定）通常会有保护机制以避免写错内存地址而危害到引导加载程序。在top-boot形式的flash装置中，受到保护的区域是装置寻址范围的顶端，而在采用bottom-boot形式的flash装置中，受到保护的区域是装置寻址范围的底部。

尽管图2-5展示的存储装置被分成4个不同的部分，不过它实际上可能没有包含那么多

译注3： 这是指多重引导的状态。

部分。引导参数可能会位于保留给引导加载程序的空间。内核也可能位于根文件系统之上，然而，这样的话需要引导加载程序具备读取根文件系统的能力。同时，内核和根文件系统也可以被包装成单一映像，使用前只要在RAM中解开该映像即可。因引导加载程序的能力的不同，甚至会有其他可能的配置，当然每种配置各有其优缺点。我们通常会使用以下的标准对设置进行分类：flash的使用、RAM的使用、更新的容易度以及引导时间。

引导存储媒体最初的编程使用装置编程器，或CPU内建的调试功能，例如JTAG或BDM。一旦装置完成了最初的编程，系统设计者就可以用引导加载程序（如果它具备此能力的话）或Linux的MTD子系统做进一步的编程。此系统还可能包含让用户轻松更新存储装置的软件。我们将会第七章探讨固态存储装置的编程方式。

磁盘

这种设置或许是你最熟悉的做法，因为它广泛应用在工作站及服务器中。此时，内核和根文件系统位于磁盘装置上。最初的内核加载程序不是从磁盘加载第二个内核加载程序，就是直接从磁盘获得内核本身。然后，磁盘上会有一个被用作根文件系统的文件系统。

开发期间，如果想要测试大量的内核及根文件系统配置，那么这种设置特别具有吸引力。例如，如果准备用定制的主流发行套件来开发嵌入式系统，那么这种设置相当有帮助。如果在产品中使用硬盘，或是模仿硬盘的装置，例如CompactFlash，这种引导方式或许是最佳的选择。

因为大家都知道这种方案，并且已经有不少的说明文件，我将只会在第九章概略提到。

网络

在这种设置中，不是根文件系统就是内核与根文件系统会通过网络连接加载。第一种情况是指，内核位于固态存储媒体或磁盘上，通过NFS安装根文件系统。第二种情况是指，只有内核加载程序位于当地的存储媒体上。然后，通过TFTP下载内核，通过NFS安装根文件系统。要自动获得TFTP服务器的位置，引导加载程序可能还得使用BOOTP/DHCP。如果使用了BOOTP/DHCP，目标板就不必为了要找到TFTP服务器或NFS服务器而事先设定IP地址。

这种设置适合应用在开发初期，因为它让开发者能够在自己的工作站与目标板间快速地共享数据和软件，而不必编程目标板。相对来说，系统变成产品之后，很少再使用这种设置，因为它需要提供服务的服务器。不过，在第一章提到的控制系统中，这种设置确实可以用在某些装置上，因为SYSM模块已经提供网络服务了。

显然，这种设置涉及了设定服务器，使其提供适当的网络服务。我们将会在第9章探讨如何为这些网络设定配置。

系统存储器的设计

要想善用可用资源，就必须了解系统内存的设计，以及物理地址空间与内核的虚拟地址空间有何不同（注4）。最重要的是，许多硬件外设可以在系统的物理地址空间进行存取，但是限制在虚拟存储空间中存取或者完全“不可见”。

为了解释虚拟和物理地址空间的差异，让我们进一步检查范例系统的某个组件。例如，用户接口模块，可以轻易在使用 StrongARM 的 iPAQ PDA 上实现。图 2-6 展示了执行 Familiar 发行套件的 iPAQ 的物理和虚拟存储器配置图。请注意，图中每个区域的划分未必与其在内存中的实际大小成比例。真要这么做，有许多区域都会小到看不清楚。

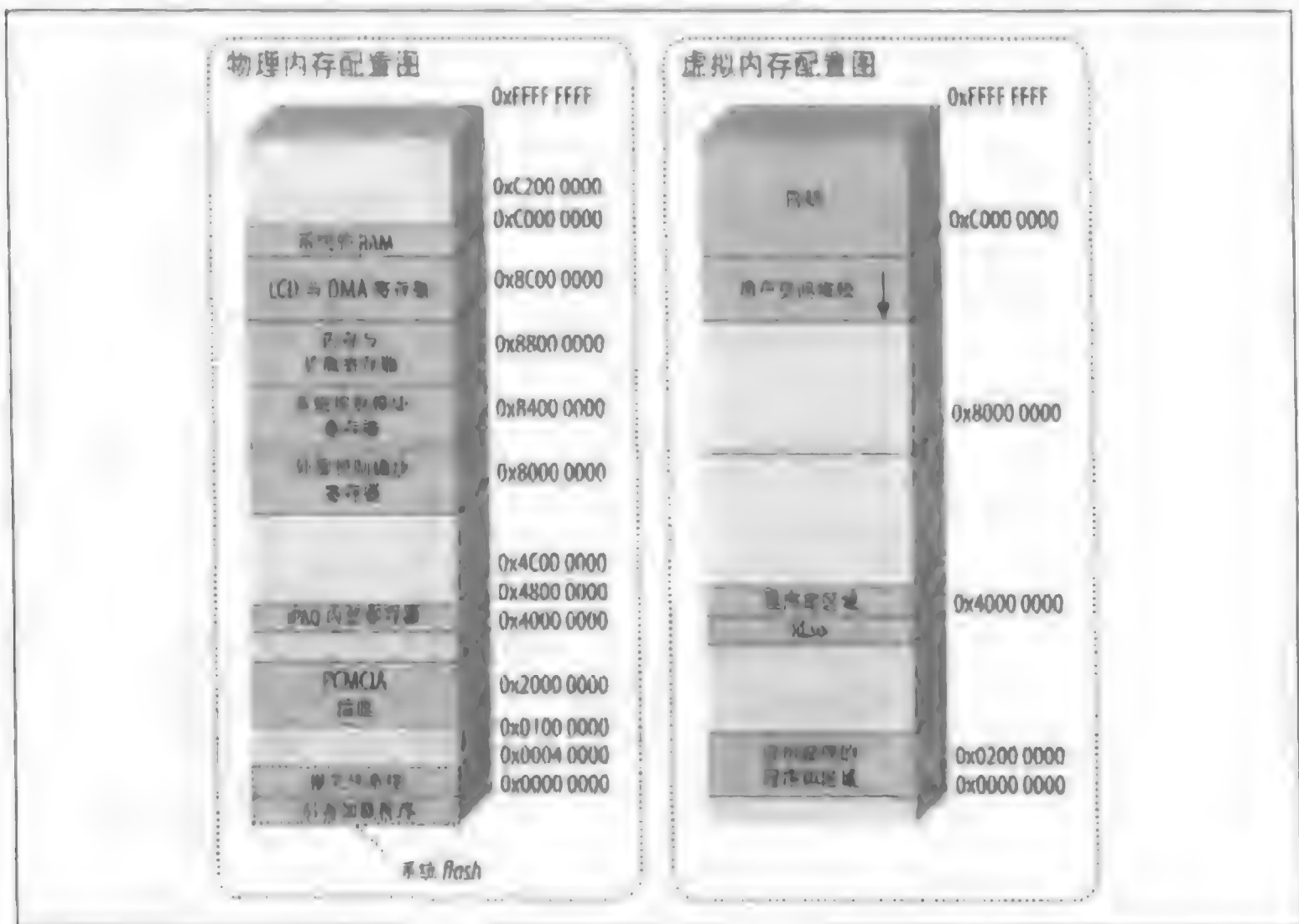


图 2-6: Compaq iPAQ 的物理和虚拟内存配置图

注4：此处所说的“虚拟地址”（virtual address）用 x86 架构的术语称为“逻辑地址”（logical address）。在其他架构上也可能会有其他的用语。

系统的物理存储器配置图通常可以在硬件附带的技术文献上看到。以 iPAQ 为例，可以到 Intel 的网站获得 StrongARM 手册《SA-1110 Developer's manual》。

物理存储器配置图之所以重要，是因为它可以提供如何设定内核配置及如何开发定制驱动程序的信息。例如，在设定内核配置期间，可能需要指定 flash 装置位于系统何处。在开发期间，也可能需要为存储器映射的外设撰写驱动程序。此外，还需要为引导加载程序提供所加载组件的相关信息。因此，最好的做法是，在你进行软件开发之前，先花些时间建立好系统的物理存储器配置图。

在 iPAQ 上，flash 存储空间被分成两个部分。第一个部分包括引导加载程序，从最低的物理地址开始。就引导加载程序的实际大小来说，所能画出的区域实在很小。flash 存储空间的其余部分被系统的根文件系统占用，倘若系统使用的是 Familiar 发行套件，也就是 JFFS2 文件系统，在这种情况下，内核实际上就位于根文件系统。这是有可能的，因为引导加载程序对 JFFS2 有充分的了解，所以能够在文件系统上找到内核。

启动之后，引导加载程序会立即从根文件系统将内核读进系统的 RAM，而且会跳到内核的启动例程执行。从此之后，系统启动过程的其余部分都交给 Linux 来完成。

一旦 Linux 运行之后（注 5），程序使用的是虚拟地址。与物理存储器配置图相比，虚拟存储器配置图的安排是内核配置设定或设备驱动程序开发的第二个重点。例如，开发设备驱动程序应该充分了解哪些信息位于内核空间，哪些信息位于用户空间，以及必须使用哪些函数才能正确交换这两个空间的数据。

虚拟存储器的配置之所以重要，是因为它可以帮助你了解应用程序，以及对应用程序进行调试。正如你在图 2-6 中所见，内核从地址 0xC0000000 开始占据了虚拟地址四分之一的空间。这个区域就是所谓的“内核空间”。地址空间的其余部分为应用程序专属的程序代码区域、数据区域和链接库区域所占据。这就是所谓的“用户空间”。由于内核会将所有的应用程序放在地址 0xC0000000 之前，所以应用程序的内存配置，即使在相同的系统上也可能不一样。

若想重建进程的虚拟存储器配置图，可以检查 */proc* 文件系统中进程的 PID 条目所提供的 *maps* 文件。至于如何获得此信息的进一步细节，请参考《Understanding the Linux Kernel》（O'Reilly）第二十章。

注 5： 此处假定你使用的是配备 MMU 的硬件。如果使用的是没有 MMU 的处理器，此处的说明就不适用了。

第三章

所支持的硬件



介绍过嵌入式Linux系统的基本概念以及一般系统架构，现在让我们来探讨Linux支持的嵌入式硬件。我首先会介绍Linux支持的处理器中常用于嵌入式系统的部分。接着，我将会介绍相关的各种硬件组件，例如总线、输出/输入设备、存储设备、通用网络设备、工业级网络设备以及系统监控设备。尽管我的说明涵盖了许多不同的组件，但是我也省略了一般不会用于嵌入式系统的组件。

请注意，接下来的讨论并不会涉及硬件组件的优缺点。当你想确定系统中包含哪些组件，或是想判断究竟要花多少功夫才能让Linux运行在自己所选的硬件上，它可以作为你研究的起点。

同时，接下来的讨论也不会涉及各种硬件厂商为了支持其硬件而提供的软件。它只会包含开放源码和自由软件社群支持的硬件。有些厂商可能会为自己的硬件提供封闭源码的驱动程序。如果真的想要使用这类硬件，不要忘了这将无法获得开放源码和自由软件社群的支持。一旦封闭源码驱动程序引起或出了任何问题，到时候都得靠厂商来处理。不管是谁，只要使用的是封闭源码驱动程序，一旦出了问题，开放源码和自由软件社群的开发者都会拒绝提供帮助。

处理器架构

Linux可以运行在大量的处理器架构上，但正如之前所说，并不是所有架构都能够实际应用。接下来我们会讨论Linux支持的CPU隶属何种架构，以及使用这些CPU构建成的目标板。同时也会包含Linux支持的其他组件，以及任何可能的警告。然而我将不会提到uClinux支持的无MMU架构。尽管此计划维护的程序代码不久前整合进2.5开发版内核系列，uClinux分支的开发仍继续进行着，而且它的周边软件也各有特色。如果想在无MMU架构上运行Linux，欢迎你到uClinux计划的网站

<http://www.uclinux.org/> 看个清楚。uClinux 目前支持 Motorola 无 MMU 的 68K 处理器、无 MMU 的 ARM、Intel 的 i960、Axis 的 Etrax，以及其他无 MMU 的处理器。

x86

Linux 对 x86 系列处理器的支持，从 Intel 于 1985 年推出的 386 开始，一直涵盖该处理器的所有后代，包括 486 和 Pentium 系列，以及其他厂商（如 AMD 和 National Semiconductor）的兼容处理器。但是，Intel 仍是 x86 系列的主要参考对象，也仍是此系列处理器的最大供货商。近来有一个新趋势，就是以 386 系列中某个处理器作为 CPU 核，再整合进传统 PC 的功能，而成为片上系统（System-on-Chip, SoC）。National Semiconductor 的 Geode 系列及 ZF Micro Devices 的 ZF x86 都是顺应 SoC 潮流的产品。

尽管 x86 是最受欢迎、最广为人知的 Linux 运行平台，但它只能代表传统嵌入式系统市场的一小部分。通常，基于复杂度和整体成本的考虑，设计者宁可选择 ARM、MIPS 和 PowerPC 处理器，也不会使用 i386。

然而，i386 仍然是使用最广泛并且测试最充分的 Linux 平台。i386 的优点是拥有 Linux 最大的软件库做后盾。许多应用程序和附加程序在移植到 Linux 支持的其他架构之前都先出现在 i386 上。事实上，Linux 的内核在移植到任何其他架构之前，就是先为 i386 写的。

因为，就功能和编程（程序设计）来说，大多数的（如果不是全部的话）i386 嵌入式系统与 i386 工作站和服务器非常类似或者完全相同，而且各种 x86 CPU 及相应目标板的 Linux 内核仅有些微差异或者根本就没有差异。需要的时候，还可以使用 `#ifdef` 语句来提供 CPU 或目标板的特性，不过这种机会很少就是了。

使用 i386 的 PC 架构应该是文档最丰富的架构，因为有许多用各种语言撰写的探讨这种架构的各类书籍和在线文档。此外，还可以从处理器厂商那里获得文档，这些文档中有些可能非常完整。如果想知道大致有哪些既有文档，可试着在 Amazon.com 上搜寻与“pc architecture”有关的书籍。要推荐一个可以涵盖 i386 和 PC 架构所有相关信息的来源实在很难。以 i386 程序设计来说，Intel 出版的《Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture》、《Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference》、《Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide》尽管内容仅局限在 Intel 的产品，但涵盖的信息却相当广泛。这三份文档的可用性或许有所不同。基于某个理由，Intel 的图书数据中心并未提供这些文件的打印版本，不过却有为这些文件提供 PDF 格式的在线版本。在写作本书时，已经可以从 Intel 的图书数据中心获得这些文档的打印版本。

至于 PC 架构本身，我找到了一个 DOS 的共享软件，叫作 HelpPC（注 1），此套件所包含的文档描述了 PC 架构各种组件的细节与操作。John Choisser 和 John Foster 合写的《The PC Handbook》（Annabooks）则是类似信息的另一个来源。实际信息请参考硬件文档。

ARM

ARM（Advanced RISC Machine 的缩写）是 ARM Holdings Ltd. 维护并推动的处理器系列。与其他芯片制造商（例如 IBM、Motorola 和 Intel）不同的是，ARM Holdings 不会制造自己的处理器。ARM 会依据 ARM 核为客户设计 CPU 核，向客户收取该设计的许可费，并让客户根据需求设计制造芯片。这么做对参与其中的各方有许多好处，但也必然会对第一次使用此架构的开发者造成混淆，开发者们会觉得 ARM 芯片在市场上好像没有中心制造商一样。然而，请记住有一点很重要，它们全都拥有一致的特性：所有的 ARM 处理器共享相同的 ARM 指令集，这让不同版本的 ARM 处理器能够达到软件上的完全兼容。但这并不代表所有 ARM CPU 和目标板的编程和设置方法都一样，对所有的 ARM 处理器来说，只有汇编语言及其产生的二进制代码一样。目前，ARM CPU 的制造商包括 Intel、Toshiba、Samsung 以及许多其他厂商。ARM 架构在许多应用中都非常受欢迎，而且有上百家厂商为它提供产品与服务。

撰写本书时，Linux 支持 10 种 ARM CPU、16 种平台以及 200 种目标板。由于篇幅有限无法详述这些信息，想要了解 Linux 支持哪些 ARM 系统，最新最完整的信息都可以在 <http://www.arm.linux.org.uk/developer/machines/> 找到。总之 Linux 一定会支持最主流的 CPU 和目标板就是了，如 Intel 的 SA1110 StrongARM CPU 以及 Assabet 的开发板。如果想要提供 Linux 支持的新硬件也可以通过上面这个网址。一般来说，如果要知道与 Linux ARM 移植有关的任何信息，查询 <http://www.arm.linux.org.uk/> 提供的相关计划的网站就对了。

除了将 Linux 内核移植到 ARM 的计划，还有许多计划对 ARM 的支持已经做好了准备。首先，RTAI 计划提供的硬实时支持以及对 StrongARM RTLinux 的移植已经可以从 <http://www.imec.be/rtlinux/> 获得。此外，对 Java 的支持也可以从 Blackdown 计划获得

注 1：只要搜寻与“HelpPC”有关的网站，应该可以立即找到此套件的下载网址。尽管该共享软件中的文档内含特殊格式，可供 HelpPC 这个 DOS 公用程序读取，不过这些文档本身却是一般的文字文档，可以使用 Linux 上任何编辑器进行读取。

(注2)。然而，由于没有内核调试器，大多数需要在ARM系统上进行内核调试的开发者都会使用JTAG调试器。

与ARM架构有关的任何信息及其指令集，可参阅David Seal编写的《ARM Architecture Reference Manual》(Addison-Wesley)以及Steve Furber的《ARM System-on-Chip Architecture》(Addison-Wesley)。与其他厂商相比，ARM并没有为自己设计的芯片提供免费手册。目前就英文版来说，ARM只有这些参考手册，尽管《ARM Architecture Reference Manual》的内容不如其他处理器厂商提供的技术手册成熟，不过已经够用了。这是因为负责芯片开发工作的芯片制造商，可以提供与其产品有关的特殊信息，如时序和机械性能方面的数据。例如，Intel为它的StrongARM实现提供的可在线读取的手册。

IBM/Motorola PowerPC

PowerPC架构是IBM、Motorola和Apple共同合作的成果。PowerPC继承了这三家公司的技术，尤其是IBM的Performance Optimization With Enhanced RISC (POWER)架构。PowerPC主要是因“使用在Apple的Mac计算机上”而闻名，不过IBM和其他厂商的产品中也可以看到使用PowerPC的工作站，此外你还可以看到使用PowerPC的嵌入式系统。例如，受欢迎的TiVo系统就内嵌有PowerPC处理器。

跟i386和ARM一样，Linux对PowerPC (PPC)架构支持得非常好。支持的程度从Linux运行在大量的PPC CPU和系统上来可见一斑。

为了兼容各种PPC硬件，每种PPC架构的底层功能会被归类放到相应的文档中，而且这些文档都会标明所属的架构。例如，那些与CHRP、Gemini和PReP机器相应的文档(译注1)。这些文档的名称反映出了其所属的架构，如`chrp_pci.c`或`gemini_pci.c`。PPC的嵌入式版本内核也是同样的道理，例如IBM的4xx系列以及Motorola的8xx系列。

此外，i386上的大量应用程序也可供PPC使用。例如，RTLinux和RTAI便支持PPC。另外，Java和OpenOffice也已经移植到了PPC上。PPC Linux社群活跃在许多开发领域中，范围从工作站到嵌入式系统都有。主要的PPC Linux网站位于<http://penguinppc.org/>。此网站由社群成员自己维护，并不附属任何特定厂商。网站上包含有用的文件和

注2: Blackdown计划是Linux的主要Java实现，它的网址在<http://www.blackdown.org/>。如果要评估特定架构对Java的支持到什么程度，我会以Blackdown计划所提供的Java执行期环境套件对该架构的支持程度作为主要的参考依据。有的架构可能只有商业的Java解决方案，不过只要不是开放源码的方案，本书都会略过不谈。第四章将会深入探讨Linux对Java的支持。

译注1: 以2.4版的linux内核为例，这些文档就放在`/usr/src/linux-2.4/arch/ppc/platforms`目录中。

链接。任何人想要在 PPC 上进行 Linux 的开发，都应该以此处为起点。还有一个与 LinuxPPC 发行套件有关的网站 <http://www.linuxppc.org/>。LinuxPPC 的移植成果最初就放在该网站。当我们在探讨发行套件的时候，值得注意的是有些发行套件只支持 PPC。例如，LinuxPPC 和 Yellow Dog Linux 提供的发行套件只支持 PPC 机器。不过传统的主流发行套件，包括 Mandrake、Debian 和 SuSE，对 PPC 的支持只是它所支持的其中一个架构。

如果打算在自己的嵌入式应用中使用 PPC，并且想要跟其他在自己的系统中也使用此架构的任何人接触，别忘了订阅讨论热烈的 linuxppc-embedded 邮件论坛。有许多问题会重复出现在该论坛中，因此或许已经有人在你之前问过相同的问题。如果没有，许多人也会关心问题是否获得了解决，因为他们也可能会遇到相同的问题。此邮件论坛以前的讨论内容收藏在 <http://lists.linuxppc.org>，该网站还包含许多其他与 PPC 有关的邮件论坛。

支持 PPC 的公司一致同意使用标准系统架构来开发使用该芯片的目标板。他们最初使用的是 PowerPC Reference Platform (PowerPC 参考平台, PReP)，此架构后来被 Common Hardware Reference Platform (通用硬件参考平台, CHRP) 取代。CHRP 方面的文件有《PowerPC Microprocessor Common Hardware Reference Platform: A System Architecture guide》(Morgan Kaufmann Publishers) 这本书。欲了解 IBM 和 Motorola 的 32 位 PowerPC 可参考《PowerPC Microprocessor Family: Programming Environments for 32-bit Microprocessors》这份手册。你可以到这两家公司的网站的技术文档专区阅读这份手册，或者到 Motorola 网站的文献中心专区索取免费的打印稿。

MIPS

MIPS 是 John Hennessey 的心血结晶，世界各地的计算机科学系学生大都知道他与 David Patterson 合著的计算机架构方面的书籍，MIPS 也是史丹福大学的 Microprocessor without Interlocked Pipeline Stages 计划的研究成果。著名的 MIPS 应用包括 SGI 销售的工作站和服务器的，以及 Nintendo 的 64 位系统和 Sony Playstations 1、2 等游戏机。与 ARM 非常类似，MIPS 由控制它的 MIPS Technologies Inc. 公司为第三方颁发 CPU 内核许可。然而，与 ARM 不同的是，MIPS 具有许多不同的指令集实现。32 位 MIPS 的实现厂商包括 IDT、Toshiba、Alchemy 和 LSI。而 64 位 MIPS 的实现厂商则包括 IDT、LSI、NEC、QED、SandCraft 和 Toshiba。

Linux 对 MIPS 的移植最初的重点放在工作站上。后来也涵盖了开发板和嵌入式系统。为了满足以 MIPS 为基础的各种 CPU 和系统的需要，在内核源码树中 MIPS 部分的安排是根据内核将会执行的系统类型分不同的目录摆放。至于 MIPS 芯片的实际类型反倒不是很重要。

与其他架构（例如 i386 或 PowerPC）相比，Linux 对 MIPS 的支持似乎很有限。事实上，只有少数的主要发行套件被移植到 MIPS。商业厂商对 MIPS 的支持似乎局限在嵌入式架构。Debian 的移植包括大端和小端形式的 MIPS，还可以看到 Red Hat 7.1 的移植。此外，许多 PDA 和开发板的制造商还主动为他们使用 MIPS 的硬件移植 Linux。如同其他架构的移植，MIPS 也缺乏适当的 Java 支持。RTAI 计划对某些 MIPS 目标板提供了硬实时支持。

除了传统上使用 MIPS 的机器，还可以看到使用 NEC 的 VR 芯片来运行 Windows CE 的机器。有些开发者为了让 Linux 支持这类架构而发起新的计划。这类计划进展很快，马上就成为了 Linux-on-MIPS 的主要开发计划。

与 Linux 的 MIPS 移植有关的进一步信息，可参考 The home of the Linux MIPS port 位于 <http://www.linux-mips.org/> 的官方网站。该网站提供了所支持的硬件、文档、相关链接，以及其他有用的资源。与 VR 和 MIPS 在 PDA 上的成果有关的信息，可参考 Linux VR 位于 <http://linux-vr.org/> 的网站。如果想了解如何在 MIPS 架构上使用 Linux，建议参考上面提到的这两个网站。此外，还可以看到商业性质的嵌入式发行套件对某些 MIPS 目标板提供广泛的支持。想在 MIPS 系统上运行 Linux，实际采用的方法将取决于所选用的目标板和开发模型。

因为 MIPS 可以划分成多种平台，必须参考系统制造商提供的数据来评估和（或）落实 Linux 的支持。就一般的资源来说，MIPS Technologies Inc. 的网站建议阅读 Dominic Sweetman 的著作《See MIPS run》（Morgan Kaufmann Publishers）。你还可以到 MIPS Technologies Inc. 的网站获得 PDF 文件。MIPS Technologies Inc. 对其一系列共三册的《MIPS Architecture for Programmers》文档提供了 32 位和 64 位的版本：《Volume I: Introduction to the MIPS Architecture》、《Volume II: The MIPS instruction set》和《Volume III: The MIPS Privileged Resource Architecture》。

Hitachi SuperH

为了加强 8 位和 16 位的 H8 微处理器产品线，Hitachi 推出了 SuperH 处理器产品线。SuperH 产品线的内部使用 32 位的数据总线，而外部则使用各种宽度的总线。后来，Hitachi 协同 STMicroelectronics（以前的 SGS-Thomson Microelectronics）成立了 SuperH Inc.。SuperH Inc. 许可和推行 SuperH 的做法与 ARM Holdings Ltd. 推行 ARM 以及 MIPS Technologies Inc. 推行 MIPS 的方式非常类似。SuperH 的早期实现，例如 SH-1、SH-2 以及其变体，没有包含 MMU。然而，自从 SH-3 开始，所有的 SuperH 处理器都包含了 MMU。SuperH 用于 Hitachi 自己的产品上、许多消费性的嵌入式系统（例如 PDA）上，以及 Sega 的 Saturn 和 Dreamcast 游戏机上。

因为早期的 SuperH (SH) 处理器没有 MMU, 所以并未受到 Linux 的支持。目前, 虽然 SH-3 和 SH-4 受到 Linux 的支持, 但是并未涵盖所有的 SH-3 和 SH-4 系统, 因为它们存在着性能各异的许多变体。Linux 支持 7707、7708、7709 等 SH-3 处理器, 以及 7750、7751、ST40 等 SH-4 处理器。因此, Linux 也支持使用这些处理器的一些系统, 包括 Sega Dreamcast。尽管已经停产, 此系统却是你熟悉非 i386 嵌入式系统的最佳实践平台。SH-5 的移植也在进行中, 但尚未成为主内核树的一部分。与此项移植有关的进一步信息, 请参考该计划位于 <http://www.superh-software.com/linux/> 的网站。

除了内核之外, SH 受到的支持, 目前还相当有限。例如, 尽管该架构具有内核调试器, 但并不支持 Java。另外, A&D Co. Ltd. 的 Masahiro Abe 还将 RTLinux 移植到了 SH-4。不过这项移植并非 FSMLabs 发行的 Open RTLinux 的一部分。你可以在 <ftp://ftp.aandd.co.jp/pub/linuxsh/rtlinux/current/> 找到该项移植。此外, 还有许多开发者为了提升 Linux 对此架构的支持程度正在努力工作中, 这包括 Debian 的移植。因此, 也可以在一些网站找到与 Linux SH 有关的文件、资源和软件。<http://linuxsh.sourceforge.net/> 和 <http://www.ml7n.org/linux-sh/> 是其中两个主要的网站。

因为没有标准的 SH 架构, 所以你必须参考与硬件有关的文档, 以了解硬件在设计和功能上的细节。此外, 还可以找到关于处理器的运行和指令集的手册。SH-3 的运行可参考《Hitachi SuperH RISC engine SH-3/SH-3E/SH3-DSP Programming Manual》, SH-4 的运行可参考《SuperH RISC engine SH-4 Programming Manual》。这两项资源都可以从 Hitachi 的网站获得。

Motorola 68000

Motorola 68000 系列在 Linux 中称为 m68k, Linux 支持其配备 MMU 的产品已经有一段时间了, 并且从 2.5 开发版系列开始支持其未配备 MMU 的产品。20 世纪 80 年代 m68k 的市场占有率仅次于 x86。m68k 除了曾被 Atari、Apple 和 Amiga 应用在许多主流的系统上, 以及曾被 HP、Sun 和 Apollo 应用在流行的工作站系统上, 它还是嵌入式系统开发选用的平台之一。然而, 近来嵌入式系统开发者的兴趣已经从 m68k 转移至较新的架构, 例如 ARM、MIPS、SH 和 PowerPC。

Linux 支持许多使用 m68k 的系统, 它起初支持先前提到的主流的系统和工作站系统, 并且包括 Motorola 的 VME 系统, 以及 BVM。这些系统互不相同, 内核源码树的结构在设计上考虑到这一点, 因此可轻易加入其他使用 m68k 的系统。每个系统都具备一组连接硬件的专用模块, 例如中断向量表与相应的处理函数。每个系统处理这些模块的方法各不相同, 内核源码中的相应函数也不同。

因为 m68k 配备 MMU 的版本使用了当今很少用到的尖端设计，所以软件方面的支持就落后了。例如，它既不支持硬实时功能，也不支持 Java。它也不在其他用户层应用程序（例如 OpenOffice）的支持清单中。与移植、所支持的硬件有关的最新信息以及相关的资源可参考位于 <http://www.linux-m68k.org/> 的 m68k Linux port 首页。Debian 发行套件对 m68k 的移植做了很多努力，如果打算部署基于 m68k 的嵌入式 Linux 系统，可以参考 Debian 的文件和邮件论坛。

因为使用 m68k 的系统没有像 i386 架构的 PC 那样的标准平台，所以单一的资源无法涵盖所有使用 m68k 的系统。然而，不难找到探讨 m68k 传统用法和程序设计的教科书和在线资源。你可以免费获得 Motorola 提供的《68000 Family Programmer's Reference Manual》（译注 2）和《M68000 8-/16-/32-Bit Microprocessors User's Manual》（译注 3）。此外，如果想阅读提供详细的范例和应用的书籍，可以到任何的网络书店以“68000”关键词进行搜寻。

总线与接口

总线与接口是系统中把 CPU 连接至外设的设备。每种总线与接口各有其复杂之处，Linux 对不同的总线与接口的支持程度将因其复杂度而有所不同。接下来我将会扼要说明嵌入式系统中可以找到的总线与接口，以及探讨 Linux 对他们提供的支持。Linux 还支持更多的总线，例如 SBus、NuBus、TurboChannel 和 MCA，不过它们只会在工作站或服务器上出现。

ISA

Industry Standard Architecture（工业标准架构，ISA）总线是 PC-AT 架构的核心。ISA 在当时是一个奇特的总线，因为它并不支持其他总线所提供的许多能力，包括轻易映射至处理器物理寻址空间的能力。然而，由于它的架构简单，所以 PC 上许多设备都广泛采用这种总线，因此在嵌入式应用中使用 PC 也是同样的情况。

存取 ISA 设备主要是通过对 I/O 端口进行编程实现的，x86 的指令集里已经包含了相关的命令。因此，要让设备驱动程序使用总线，内核并不需要特别做什么事情。设备驱动程序可以直接使用 in/out 汇编语言指令来存取适当的 I/O 端口。尽管内核支持即插即用设备，不过嵌入式应用很少使用这种能力。倒是嵌入式系统需要支持具热插拔功能（在系统执行期间新增或移除硬件）的总线，如 CompactPCI、PCMCIA 和 USB。尽管内核

译注 2: http://e-www.motorola.com/files/archives/doc/ref_manual/M68000PRM.pdf。

译注 3: http://e-www.motorola.com/files/32bit/doc/ref_manual/MC68000UM.pdf。

还支持扩展ISA (Extended ISA, EISA) 设备, 但是这种总线使用得并不是很普遍, 而且已经被 PCI 总线取代。

ISA 总线的相关信息可以在许多地方找到。之前提到的《The PC Handbook》和 HelpPC 就是关于端口编号及其操作的非常好的快速参考信息。而 Anderson 与 Shanley 合著的《ISA System Architecture》(Addison-Wesley) 则对 ISA 总线及相关硬件提供了深入的说明。此外, Rubini 与 Corbet 合著的《Linux Device Drivers》(O'Reilly) 则包含了如何在 Linux 进行 ISA 程序设计的细节。

PCI

由 PCI Special Interest Group (PCI 专责小组, PCI-SIG) 制订的 Peripheral Component Interconnect (外围组件互连, PCI) 总线可说是目前使用的最普遍的总线。PCI 被设计来取代 ISA, 它可以搭配不同的处理器架构 (含 PPC 和 MIPS) 来建立不同的系统类型, 包括嵌入式设备。

与 ISA 不同的是, 设备驱动程序必须通过软件的支持才能使用 PCI。此项支持的第一个部分包括, 在引导期间初始化并配置 PCI 设备。在 PC 系统上, 这项工作传统上由 BIOS 来完成。然而, 内核本身也具备执行此项工作的能力。如果由 BIOS 来执行初始化的工作, 则内核将会浏览 BIOS 的数据表以取回 PCI 信息。不管是由 BIOS 还是内核来进行初始化的工作, 内核都可以让设备驱动程序通过 API 获得 PCI 总线上各个设备的相关信息, 并且启动这些设备。此外用户还可以通过一些工具来操作 PCI 设备。总之, Linux 对 PCI 的支持相当完整, 并且非常成熟。

在相关的出版物资源中, 《Linux Device Drivers》一书对在 Linux 上的开发 PCI 以及操作 PCI 总线的一般方法有非常好的论述。而 Shanely 与 Anderson 合著的《PCI System Architecture》(Addison-Wesley) 对软件开发者提供了 PCI 总线的详细信息。你当然也可以从 PCI-SIG 获得正式的 PCI 规格书。不过, 正式的规格书相当枯燥乏味难以阅读。最后, 可以到位于 <http://www.tldp.org/> 的 *Linux Documentation Project* (Linux 文档计划, LDP) 网站获得《Linux PCI-HOWTO》, 该文件探讨了 Linux 使用某些 PCI 设备的注意事项, 以及 Linux 支持了哪些 PCI 设备。

PCMCIA

Personal Computer Memory Card International Association (国际个人计算机存储卡协会, PCMCIA) 既是一个总线的通称, 也是推动和维护相关标准的组织。PCMCIA 最初的标准只支持 16 位的卡片, 后来陆续推出了其他标准, 包括 32 位的 CardBus 和使用 USB 的 CardBay 规格。PCMCIA 可以让嵌入式系统更具弹性并且容易扩充。以 iPAQ 为例,

它让用户能够使用无线网络卡连上局域网。有的系统则会使用CompactFlash卡片来获得大型的永久性存储空间。

Linux 对 PCMCIA 支持的程度可能会令人感到困惑。首先，主要的 Linux PCMCIA 计划在 SourceForge 网站上，它的网址为 <http://pcmcia-cs.sourceforge.net/>，维护者为 David Hinds。此计划开发出来的套件支持大量卡片，<http://pcmcia-cs.sourceforge.net/ftp/SUPPORTED.CARDS> 备有完整的清单。Linux PCMCIA 对 i386 架构的支持相当成熟，并包括部分的 PPC 架构，遗憾的是，写作本书时它对其他芯片的支持仍处于萌芽阶段。除了 Hinds 维护的套件，正式的内核源码也支持 Hinds 的套件支持的 PCMCIA 卡片中的一部分。开发者们想让正式的内核源码成为 PCMCIA 支持的主要来源。但直到写作本书时，作为产品的系统仍然把 Hinds 的套件作为最佳选择。因为它包括了必要的系统工具，可用来配置当卡片从 PCMCIA 槽插入或拔出时，自动加载或卸载适当的 PCMCIA 设备驱动程序。

除了 PCMCIA 协会本身提供的 PC Card Standard 文档，市面上还可以看到关于 PCMCIA 的书籍。然而，在你调查有哪些参考书籍之前，应该先看一下 PCMCIA 计划网站上由 Hinds 撰写的《Linux PCMCIA Programmer's Guide》。这份指南列出了可以获得 PCMCIA 进一步信息的参考书籍。

PC/104

尽管简单，但 ISA 总线并不适合用于部署在严酷环境中的嵌入式系统。PC/104 出现的原因在于解决 ISA 机械规格的缺点。PC/104 总线提供的电气信号跟 ISA 总线一样，但前者的机械规格不仅易于扩充且坚固耐用，较适合应用在嵌入式系统的开发。与使用插槽的一般的 ISA 规格不同，PC/104 规格使用的是针脚接头。当 PCI 普及之后，PC/104+ 规格被制订出来以便提供与 PCI 信号兼容的总线，这就是 PC/104 规格的附加功能。PC/104 和 PC/104+ 由 PC/104 协会制定，该组织的成员超过 100 家公司。

从信号以及软件的观点来看，PC/104 如同 ISA，PC/104+ 如同 ISA 和 PCI。因此，Linux 支持这些总线并不需要特殊的功能。然而，这并不代表 Linux 支持所有的 PC/104 和 PC/104+ 设备。如同任何其他的 ISA 或 PCI 设备，应该搜索 Linux 是否与所评估的 PC/104 设备兼容的确切信息。

VME

VME (注 3) 总线的设计主要是依据 Motorola 于 1979 年特别为 68000 开发的 VERSA 背

注 3： 尽管按官方说法，VME 这几个字母没有任何意义，但据一位曾参与这三家公司商讨过程的工程师表示，这三个字母其实是 VERSA Module Eurocard 的缩写。

板总线。在那时，能够和 VERSA 竞争的总线包括 Multibus、STD、S-100 和 Q-bus，尽管现在已经很少使用 VERSA 了。VME 是 Motorola、Mostek 和 Signetics 合作开发的与处理器无关的总线。VME 总线使用了 VERSA 的电气信号以及 Eurocard 的机械结构。以 Eurocard 的结构来说，VME 板可垂直插入 VME 机架，并且使用针脚接头连接至其背板。这和使用边缘接头（俗称金手指）插入槽位的一般计算机板不同。自从 VME 总线出现之后，它已经成为功能强大、坚固耐用计算机最优选的总线。让 VME 总线广受欢迎的一个因素是，它是个并非受任何单一组织支配的开放式标准。

VME 总线适用多种 VME 板，而每张 VME 板都拥有各自的 CPU 和操作系统。总线并非由中央操作系统控制，而是通过仲裁机制让 VME 板暂时成为总线主控者，以执行其操作。因此，在 VME 板上的 Linux 任务必须正确地与 VME 硬件接口交互，以便获得适当的性能。

目前有两个活跃的 Linux VME 计划。第一个计划的目的在于对使用 Motorola 68K 的 VME 板提供 Linux 支持，该计划的网站位于 <http://www.sleepie.demon.co.uk/linuxvme/>。尽管该计划的大部分工作已经整合到主内核树，不过此计划的网站仍旧是你获得最新开发信息的主要资源。第二个计划与处理器无关，其目的在于对所有的 VME 板提供 Linux 的支持。该计划的名称为 VMELinux Project，其网站位于 <http://www.vmelinux.org/>。这两个计划的网站上都列有所支持的 VME 板清单。所以，当你想评估自己的 VME 板是否受到 Linux 的支持时，应该参考这份清单。如果选用的 VME 板尚未受到支持，受支持的其他 VME 板对于理解如何在 Linux 中支持 VME 板有帮助。

除了这两个计划，一些软件和硬件厂商还在自己的发行套件中对额外的 VME 硬件提供 Linux 的支持。例如，DENX Software Engineering 维护的内核对 VMELinux 计划不支持的各种使用 PPC 的目标板提供了支持，并且可以通过 CVS 到他们的网站获得这种内核。

若想从 Linux 的观点来看 VME，可以从 LDP 的网站获得《Linux VME HOWTO》文档。VMEbus International Trade Association (VITA) 网站（译注 4）推荐了一些与 VME 总线有关的出版物以及相关的标准。John Black 的《The Systems Engineer's Handbook: A guide to building VMEbus and VXibus systems》并未列在推荐清单之中，不过值得一读。

CompactPCI

CompactPCI 规格创始于 Ziatech，并由 PCI Industrial Computer Manufacturer's Group

译注 4: <http://www.vita.com/>。

(PCI工业计算机制造商组织, PICMG) 继续延伸发展, 以及推广 CompactPCI 的应用。CompactPCI 规格为高效、高可用应用提供了开放和多用途的平台。CompactPCI 成功之处主要是它基于设计者选择的技术。首先, 他们选择使用因 VME 而普及的 Eurocard 结构。其次, 他们选择与 PCI 总线兼容, 这让 CompactPCI 板制造商能够使用市场上主流的低价 PCI 芯片。

从技术上来说, CompactPCI 总线的电气信号与 PCI 总线一致。它并不使用大多数工作站和服务器的插槽, 而是使用针脚接头垂直插入 CompactPCI 背板, 这一点跟 VME 非常像。如同 PCI 一般, CompactPCI 需要单一的“总线主控者”(注 4), 相对而言, 正如前文所说, VME 容许多个总线主控者存在。因此, 在系统的槽位上一定要存在一张 CompactPCI 卡片(板), 由它负责 CompactPCI 背板使用权的仲裁, 这就好像在工作站或服务器中, 由一个 PCI 芯片组负责 PCI 总线的仲裁一样。

此外, CompactPCI 还允许实现热插拔规范(该规范描述了运行期间插拔 CompactPCI 卡片的方法与程序)。热插拔规范对热插拔的定义分三个等级。每个等级代表一组硬件和软件的能力。以下是这三个等级的说明:

基本热插拔

这个热插拔等级包括系统操作员通过控制台介入操作程序。当一张新卡片插入时, 操作员必须通知操作系统启动它, 然后设定和通知软件它的存在。移除一张卡片时, 操作员必须通知操作系统该卡片将被移除。操作系统必须停止正在与该卡片进行交互的工作, 并通知该卡片关机。

完整热插拔

与基本热插拔相比, 完整热插拔并不需要操作员通过控制台介入操作程序。取而代之的是, 操作员以拨动卡片面板上的微动开关来通知操作系统该张卡片将被移除。操作系统接着会执行必要的操作来隔离该张卡片, 并通知它关机。最后, 当这张卡片被移除之后, 操作系统会点亮一个 LED 来通知操作员。插入一张卡片时, 当操作系统收到正确的插入信号, 便会进行相反的操作。

高可用

在这个等级中, CompactPCI 卡片完全受软件控制。由热插拔控制器软件管理系统中所有卡片的状态, 并根据系统的状态处理个别的卡片。举例来说, 如果某张卡片发生故障, 控制器可以将它关机, 并启动同一个机架中另一张备用卡片。这个热插

注 4: “总线主控者”在不同的环境中可能代表不同的东西。在这个特例中, “总线主控者”专指设置及设定 PCI 总线的设备。在 PCI 总线上只能有一个这样的设备, 尽管 PCI 总线上可能有多个设备可以存取到其他 PCI 设备所用到的存储空间。

拔等级之所以称为“高可用”，是因为这个功能非常适合使用在高可用的应用中（注5），例如电信系统的停机时间必须越短越好。

通过已经提供的PCI支持，Linux能够符合基本的CompactPCI规格。不过设备的热插拔功能在Linux中以不同的形式存在。基本上，2.4版的内核包含了必要的内核功能。而相关的用户工具则可到位于<http://linux-hotplug.sourceforge.net/>的Linux Hotplugging的计划获得。

有人说这种层次的支持不足以涵盖CompactPCI系统的所有功能。此外，主要的内核源码树中仅包含少数主流CompactPCI板的驱动程序，不过CompactPCI板制造商自己可以提供Linux驱动程序。因此，CompactPCI上的高可用Linux解决方案出现了一些商业的解决方案，例如Availix的HA Cluster以及MontaVista的High Availability Framework。持续发展的High-Availability Linux Project计划，是由<http://linux-ha.org/>创建，主要目的是为那些采用Linux建立高可用的解决方案提供开放源码组件。此计划并不限于特定的硬件配置，因此它并不是以CompactPCI为中心来发展的。

未来我们可能会看到更多开放源码软件提供CompactPCI系统的各种功能，包括热插拔能力，以及支持通信、资源监控、群集管理的软件，以及在高可用系统中可以找到的软件组件。然而，现在如果想在“以CompactPCI为基础的高可用应用”中使用Linux，可能需要寻求既有的商业解决方案，以获得CompactPCI规格的所有功能。

你可以在Linux Hotplugging计划的网站上找到与Linux的热插拔能力有关的文档，包括如何编写具有热插拔能力的驱动程序以及如何设定热插拔管理工具。该网站还提供了若干相关信息的链接。与CompactPCI规格有关的信息可以向PICMG购买。

并口

尽管一般人并不认为并口是个总线，但并口却是许多计算机的基本配备，而且可被用来连接大量的外围设备，包括硬盘、扫描仪、甚至网卡。Linux对并口设备的支持相当广泛，包括内核里可以找到的驱动程序以及支持计划提供的驱动程序。然而，并不存在一个中央权责单位或计划负责Linux对并口设备的支持，因为并口是计算机系统的基本组件。不过，倒是可以找到一些“描述哪些设备受到支持”的有用资源。这包括LDP上可以找到的《Hardware Compatibility HOWTO》以及<http://www.torque.net/linux-pp.html>

注5： 为避免混淆，我将会称这个“热插拔等级”为“高可用热插拔等级”，并继续使用“高可用”来形容需要提供高可用功能的应用和系统，而不管它们是否使用CompactPCI或实现“高可用热插拔等级”。

上可以找到的《Linux Parallel Port Home Page》。值得注意的是 Linux 支持 IEEE1284 标准，此标准定义了并口与外部设备间通信的实现方法。

除了连接外部设备，并口还可以有许多用途，当我在“I/O”一节说明如何把并口作为 I/O 接口时，将会探讨并口的程序设计。

SCSI

Small Computer Systems Interface（小型计算机系统接口，SCSI）创建自 Shugart Associates，最后它在一群标准团体（包括 ANSI、ITIC、NCITS 和 T10）的发展与维护之下逐渐成为一系列的标准。尽管大多数人都认为 SCSI 是个高端工作站和服务器的接口，不过它其实是一个通用接口，可用来连接各种外围硬件。然而，只有小部分嵌入式系统会使用 SCSI 设备。这类系统通常是高端的嵌入式系统，如前面提到的以 CompactPCI 为基础的高可用系统。支持的方法就是将 CompactPCI SCSI 控制器插入 CompactPCI 背板，以此提供连接 SCSI 设备的接口。

如果想在嵌入式系统中使用 SCSI，不要忘了，尽管 Linux 支持大量 SCSI 控制器和设备，但是许多重要的内核开发者都认为内核的 SCSI 程序代码需要大改一番，或者甚至需要完全重写。但这并不表示你不应该在 Linux 上使用 SCSI 设备。这只是个警告信息，由于内核的 SCSI 程序代码在未来可能会有变动，所以计划可能会受到影响。在撰写本书时，修改 SCSI 程序代码的工作尚未开始。但可以预见的是，这类工作将会在 2.5 版内核开发期间进行。至于现在，如果想知道 Linux 支持哪些 SCSI 硬件，可以参考 LDP 的《Hardware Compatibility HOWTO》。如同并口一样，任何单一的参考资源都无法涵盖 Linux 支持 SCSI 的所有信息，因为 SCSI 接口已经是一个用户群相当大的成熟技术。

关于 SCSI 设备驱动程序架构的探讨，可以在 <http://www.torque.net/sg/> 和 <http://www.andante.org/scsi.html> 以及 LDP 的《The Linux 2.4 SCSI subsystem HOWTO》文档中找到。如果想知道如何在 Linux 上进行 SCSI 的程序设计，可以参考 LDP 的《The Linux SCSI Programming HOWTO》文档。这份文档以及 O'Reilly 出版的《Linux Device Drivers》可以说是为 Linux 开发任何 SCSI 驱动程序时的起点。由 Gary Field 与 Peter Ridge 合著的《The Book of SCSI: I/O For The Millennium》(No Starch Press) 则广泛地探讨了与 SCSI 有关的信息。如同其他标准，制订标准的团体都会提供正式的标准文件，同样地，这类文档通常都枯燥乏味难以阅读。

USB

Universal Serial Bus（通用串行总线，USB）由一群组成 USB Implementers Forum（USB 实现者论坛，USB-IF）的公司来开发和维护。最初发展 USB 的目的在于取代并口和串

口之类不完整和慢速的接口，而并口和串口是PC传统上用来连接外围设备的接口。由于具备便宜、易用和高速处理能力等特性，使得USB很快就成为用来连接外围设备的接口选择。尽管USB主要是用来连接设备的总线，不过有越来越多的嵌入式系统（如一些制造商所生产的SBC和SoC）使用它作为硬件组件。

USB设备会以树状的形式连接在一起。该树的根节点称为“根集线器”，它通常就是主板，所有的USB设备和“非根集线器”都会连接至该处。根集线器负责处理所有（直接或通过第2个集线器）连向它的设备。它的限制是，在任何形式的网络中，都无法直接使用USB缆线连接计算机（注6）。

Linux对USB根集线器（注7）的支持相当成熟而且广泛，绝对可以和支持USB的商业操作系统匹敌。尽管大多数的硬件厂商并不会对他们的USB外围设备提供Linux驱动程序，不过他们可能会借助提供硬件规格的方式来协助Linux开发者设计USB驱动程序。此外，如同其他的硬件组件，就算制造商不愿意提供相关规格，开发者依然能够设计出Linux驱动程序来。Linux通过内核中的USB堆栈来支持USB的主要组件。只要是Linux支持的USB设备，内核也会包含其驱动程序。你还可以找到用来管理USB设备的用户工具。这些用户工具以及所支持设备的完整清单可以从位于<http://www.linux-usb.org/>的Linux USB计划网站获得。

与USB根集线器相比，Linux对USB设备（注8）的支持比较有限。尽管有些运行Linux的系统，例如iPAQ，已经可以当USB设备来用，但是还没有统一的架构可以用来将USB设备功能加入Linux内核。

如果想了解USB设备的开发，可到Linux USB计划网站获得Dettlef Fliegl所著的《Programming Guide for Linux USB Device Drivers》。《Linux Device Drivers》这本书也提供了如何编写Linux USB设备驱动程序的指南。你还可以在各个网上书店找到一些探讨USB的书。然而，开发者和网络书评者一致认为，USB-IF提供的原始USB规格仍是最佳起点和最佳参考资料。

IEEE1394（火线）

FireWire是Apple对其技术所拥有的商标，此技术设计于20世纪80年代晚期90年代初

注6：事实上，有一些厂商提供了某种形式的，经由USB的主机对主机连接，但是标准并不想纳入这种形式的规划。你还可以使用USB网卡（包括Ethernet适配卡）将计算机连接到网络中。

注7：此刻，Linux负责处理连向它的所有USB设备。

注8：此刻，Linux只是另一个连向（不一定执行Linux的）USB根集线器的USB设备。

期。后来，Apple 将其成果提交 IEEE，并以此为基础制定出 IEEE 1394 标准。与 USB 非常像的是，IEEE1394 让你可以用简单且低廉的硬件接口来连接各种设备。由于其相似性，IEEE1394 和 USB 常被拿来相提并论。然而，就传输速度来看，考虑到有大量数据传输需求的设备（例如，数字相机和外接硬盘）时，显然 IEEE1394 架构要比 USB 优越许多。近来更新的 USB 标准已经缩减了它们之间的差距，不过目前对既有的高效设备和扩充能力来说，显然 IEEE1394 仍占优势。尽管只有少数的嵌入式系统使用 IEEE1394，但是当传输大量数据的需求增加时，这类技术的需求也会跟着提升。

与 USB 不同的是，IEEE1394 连接并不需要根节点。IEEE1394 连接既可以采用菊花链方式也可以使用 IEEE1394 集线器。此外，与 SCSI 不同的是，IEEE1394 连接并不需要任何终端电阻。你还可以使用 IEEE1394 直接连接两部或多部计算机，USB 就没办法这么做。为了利用这种能力，甚至有 RFC 提到如何在 IP 上实现 IEEE1394。这么做可以为具有 IEEE1394 能力的计算机提供低价且高速的网络连接。

尽管 Linux 对 IEEE1394 的支持并不如某些商业操作系统广泛，不过已经成熟到实用的程度了，可以应用在每天大量使用的 IEEE1394 硬件设备上。尽管内核源码已经包含支持 IEEE1394 的必要程序代码，不过 IEEE1394 子系统的最新程序代码以及相关的用户工具程序仍需在 IEEE1394 for Linux 计划位于 <http://www.linux1394.org/> 的网站获得。你可以在该网站的 compatibility 专区找到它支持的设备。Linux 的 IEEE1394 支持的设备数量和类型，在未来只会增多不会减少。

Linux 对 IP 上的 IEEE1394 的支持目前尚处初期阶段。等到成熟之后，这可能会变成进行嵌入式 Linux 系统调试非常有效的方法，因为主机和目标板之间可以交换大量的数据。

欲了解如何使用 Linux 底下的 IEEE1394 子系统及其支持的硬件，可以在 IEEE1394 for Linux 计划网站上找到相关文档。与 IEEE1394 相关的各种规范文件也可以在该网站找到相关链接。IEEE1394 标准本身的文档可以从 IEEE 购买，但对个人来说要花不少钱。尽管要在 IEEE1394 上进行任何扩展的工作必须参考该标准的文件，不过 Don Anderson 所著的《FireWire System Architecture》（Addison-Wesley）却是一个很好的起点。

GPIB

通用接口总线（General-Purpose Interface Bus，GPIB）就是 HP 的 HP-IB 总线。此总线诞生于 20 世纪 60 年代末期，而且目前仍旧仍然在工程以及科学领域应用。在逐渐成熟的过程中，GPIB 成为了 IEEE488 标准（译注 5），并于 1992 年二次修订（译注 6）。事

译注 5：IEEE Std 488.1-1987。

译注 6：IEEE Std 488.2。

实上，许多用于数据采集和分析的设备都配备有 GPIB 接口。每当有这方面应用的主流硬件出现，这类主流硬件，尤其是指 PC，就会有許多 GPIB 硬件适配卡可用。

你可以使用遮蔽线将 GPIB 设备连接在一起，线的两端可能是“可堆叠的”接头。接头可以堆叠在一起，指的是一条线的接头具有适当的硬件接口，可以让第二条缆线的接头附接上去，而第二条缆线的接头本身又可以让第三条缆线的接头附接上去。例如，一条将计算机连接至设备 A 的缆线，其附接设备 A 的接头又可以让另一条缆线的接头附接上去，让设备 A 连接到设备 B。

尽管内核本身并未包含任何 GPIB 适配卡的驱动程序，不过有个 Linux GPIB 计划。然而，这个计划的发展过程一波三折，它起初是 Linux Lab Project (<http://www.linux-lab.org/>) (注 9) 的一部分。经过初期开发以及发布若干个版本后（现在仍可到 <ftp://ftp.llp.fu-berlin.de/LINUX-LAB/IEEE488/> 获得），开发工作便停止了。这个套件度过了几年没人维护的日子，直到 Frank Mori Hess 最近在 <http://linux-gpib.sourceforge.net/> 重新发起此计划，并将此套件更新到 2.4.x 内核系列。此套件目前提供内核驱动程序，与 National Instruments 的 GPIB 链接库兼容的用户空间链接库，以及与 Perl 和 Python 语言的绑定。此套件支持来自 HP、Keithley、National Instruments 和其他制造商的硬件。你可以在该套件源码随附的 *devices.txt* 文档中以及该计划的网站上找到所支持硬件的完整清单。

使用此套件时，GPIB 总线在用户空间看到的是 */dev/gpib0*、*/dev/gpib1*，等等。当你为总线附接的设备设计程序时，必须知道它们的 GPIB 地址。*/etc/gpib.conf* 配置文件可以简化所附接设备地址的设定工作。必须根据配置情况对该文档进行裁剪。该套件各组件的安装与操作以及 GPIB 链接库函数的说明都可以在此套件随附的《Linux-GPIB User's Guide》文档中找到。

I²C

Inter-Integrated Circuit（集成电路间，I²C）总线最初是由 Philips 提出，目的是让电视机里的组件能够彼此通信，而今我们可以在任何规模和目的的许多嵌入式设备上发现 I²C 总线的踪影。如同其他类似的小型总线，例如 SPI（注 10）和 MicroWire，I²C 是个简单

注 9：事实上，Linux Lab Project 支持的不只是 GPIB。正如它在网站上所说，它的目标在为所有的 Linux 用户提供广泛的，可用来处理自动化、过程控制、工程和科学数据的 GPL 软件工具。

注 10：尽管 Linux 上有 SPI 的支持，但仅限于若干板子。事实上，并不存在能够提供跨架构 SPI 支持的结构。

的串行总线，可以让嵌入式系统的 I²C 组件间交换有限的数。市场上有各式各样具有 I²C 能力的设备，包括 LCD 驱动器、EEPROM、DSP 等等。因为它很简单，并且对硬件的要求不高，所以 I²C 既可以用软件也可以硬件的方式来实现。

使用 I²C 连接设备只需要两条信号线：串行时钟 (serial clock, SCL) 线和串行数据 (serial data, SDA) 线。前者用来传送时钟信号，而后者用来传送实际数据。I²C 总线上所有设备都会使用同一对信号线互连。总线上发起交易的设备会成为总线的“主控者”，它会跟寻址到的“从属者”进行通信。尽管 I²C 支持多个“主控者”，不过实现上通常只会有一个“主控者”。

主内核源码树包含对 I²C 的支持、一些使用 I²C 的设备，以及相关的 System Management Bus (系统管理总线, SMBus)。由于硬件监控传感器设备大量使用 I²C 设备，所以你可以在 Linux 硬件监控计划位于 <http://www2.lm-sensors.nu/~lm78/> 的网站找到 I²C 支持网页。此网站包含 I²C 程序代码的相关连接、文档以及最近的开发成果。最重要的是，它包含了一份所支持的 I²C 设备清单，以及每个设备使用的驱动程序。

除了内核源码附带的关于 I²C 的文档，还可以在硬件传感器网站上找到相关的连接与文档。此外，可以到 Philips 位于 <http://www.semiconductors.philips.com/buses/i2c/> 的网站获得关于总线的信息，以及相关的规格。有兴趣想要了解总线、协议及其应用的人还可以参考 Vincent Himpe 维护的 I²C FAQ (<http://www.ping.be/~ping0751/i2cfaq/i2cfaq.htm>)。

I/O

I/O 在任何计算机化设备上都扮演着重要的角色。如同其他操作系统，Linux 对 I/O 设备的支持非常广泛。接下来，我并不打算将它们全部列出来一一说明。有这种需要的人，可以阅读 LDP 提供的“Hardware Compatibility HOWTO”文档。接下来，我会将重点放在 Linux（通过内核或是通过用户应用程序）支持不同类型的 I/O 设备的方式。

这里探讨的 I/O 设备，有部分受到内核两种形式的支持：其一是原生驱动程序，用来处理直接连上系统的设备；其二是通过 USB 层，因为设备可能会附接至该处。例如，PS/2 键盘和并口打印机，以及 USB 键盘和 USB 打印机。因为前面已经探讨过 USB 了，而且篇幅有限无法深入探讨 Linux 的 USB 堆栈，所以此处将只会说明 Linux 对直接连接在系统上的设备提供的支持。然而，请注意，对于类似的设备来说，USB 驱动程序会依赖 Linux 中用来支持原生设备既有的基础结构。例如，USB 串行适配卡驱动程序依赖的是跟传统串行驱动程序一样的底层结构以及 USB 堆栈。

串口

串口可说是每个嵌入式系统开发者最好的朋友（或是最坏的敌人，取决于他过去对这个普遍存在的接口的使用经验）。许多嵌入式系统的开发和调试，使用的是主机和目标板之间的RS232串行连接。有时，尽管设计PCB的时候设计了串口，但只有用来开发的板子才会装上接头，最后卖出来的产品并不会装上此接头。简单的RS232接口受到广泛的使用和采用，尽管与其他传输方式相比，它的频宽相当有限。请注意，除了RS232，还有其他的串行接口，其中有部分对噪声较不敏感，因此多半用在工业环境中。然而，硬件串行协议的重要性不如对串行设备硬件的接口进行实际的程序设计。

因为RS232是硬件接口，所以内核不需要支持RS232本身。内核包含的是实际进行RS232通信的芯片、Universal Asynchronous Receiver-Transmitter（通用异步收发器，UART），的驱动程序。尽管UART因架构而异，然而有些UART，例如16550，却可以在多种架构上使用。

内核里，主要的串行（UART）驱动程序位于 *drivers/char/serial.c*。有些架构，例如SH，具有其他的串行驱动程序以配合其硬件。有些与架构无关的外围卡也提供串行接口。如同其他的Unix系统，串行设备在Linux中就像终端设备一样，具有一致的存取方式，而与底层的硬件和驱动程序无关。终端设备文件的范围从 */dev/ttyS0* 开始一直到 */dev/ttyS191*。然而，在系统的 */dev* 目录中通常只能找到几个串行设备文件。

你可以在LDP的“Serial HOWTO”找到串口的基础知识、设置和配置。此外，还可以在LDP的“Serial Programming HOWTO”找到如何在Linux中进行串口程序设计的知识。因为串口的程序设计实际上就是终端的程序设计，在Unix系统程序设计方面任何好的参考资源都将会是好的起点。值得注意的是Richard Stevens所著的《Advanced Programming in the UNIX Environment》，这本大家一谈到Unix系统程序设计就会想到的著作，便包含终端I/O的内容。

并口

与串口相比，并口难得会成为嵌入式系统的重要组件。除非该嵌入式系统实际上是一个PC类型的SBC，事实上并口也很难成为系统硬件的一部分。在某些情况下，之所以会使用并口，是因为嵌入式系统必须驱动打印机或某种外部设备，不过由于出现了USB和IEEE1394，用到并口的可能性越来越小。

然而，有个领域的嵌入式系统开发相当适合采用并口，那就是多位I/O。例如，进行调试的时候，只要将一组LED连接在并口的脚位上，便可使用这些LED来指示程序代码的位置。这个技巧就是在程序代码各个位置上插入一组并口的输出命令，并使用LED来

辨别机器被锁住之前运行的最后一个位置。这是可能的，因为并口的硬件会保留上一次系统输出给它的值，而且这跟系统接下来的状态无关。《Linux Device Drivers》这本书深入描述了如何用并口作为简单的I/O接口，以及如何设置LED数组来显示并口的输出。

Linux通过一组三层结构来支持并行I/O。中间层是个与架构无关的parport驱动程序。该驱动程序会为并口资源提供中央管理能力。这个中间层驱动程序，在用户空间上是看不到的，而且也没办法通过/dev目录中的设备文件来存取。用来控制实际硬件的底层驱动程序会跟中间层驱动程序注册，让高层驱动程序能够取用底层驱动程序提供的服务。而高层驱动程序则可能对外提供不同的服务。底层和中间层驱动程序可以在内核源码树中的drivers/parport目录里找到。

最常见的高层驱动程序就是line printer驱动程序，它可让用户的应用程序直接使用附接在系统并口上的打印机。在用户空间中可以看到的第一部line printer设备是/dev/lp0，第二部是/dev/lp1，以此类推。有些高层驱动程序会把并口当成是一个扩展总线来存取附接到系统上的外部设备。不管怎样，这些高层驱动程序都会使用中间层驱动程序，并且可以在/dev目录中找到相应的设备文件。最后，我们可以从用户空间以原生的并口驱动程序/dev/parportX来存取并口本身。该驱动程序位于内核源码树中的drivers/char/lpdev.c文档里。

除了前文提到的一般PC架构参考资源以及设备驱动程序书籍，还可以在<http://people.redhat.com/twaugh/parport/>提供的“The Linux 2.4 Parallel Port Subsystem”文件和内核源码树中的Documentation目录里找到关于Linux并口子系统和API的说明。

调制解调器

嵌入式系统使用调制解调器拨接数据中心是相当常见的事情。警报系统、银行柜员机以及远程监控硬件，都是嵌入式系统需要跟中央系统通信以满足其主要目的的好例子。尽管目的各不相同，但是这些系统都会以传统的调制解调器连接POTS（旧式电话系统）来访问远程主机。

调制解调器在Linux中会被当作串口，即使在不同的操作系统中，包括Unix，也几乎都是一样的情况。因此，调制解调器的存取可通过/dev目录中适当的串行设备文件来进行，而且会受到跟原生的串行UART一样的驱动程序的控制，这与调制解调器是内部或外部设备无关。然而，这项支持只能应用在真正的调制解调器上。

近来，PC市场出现了一种称为WinModem的调制解调器。WinModem只提供组成调制解调器的最起码硬件，通过在操作系统上执行软件来提供实际的调制解调器服务。正如它的名称所暗示的，这类调制解调器主要是以Windows为执行平台。它们之所以能够在

Windows 正常运行，是因为厂商为该操作系统提供了适当的驱动程序，然而，它们无法在 Linux 上使用，因为它们并不包含完整的调制解调器硬件，所以内核无法使用串行驱动程序来操作它们。

为了支持这类（有缺陷的）设备，有若干计划被发起，以便开发必要的软件套件。这些计划的中央权责单位是位于 <http://www.linmodems.org/> 的 Linmodems 网站。此网站提供各种 WinModem 支持计划的文件、消息和链接。然而，在我写作本书时，还没有人能够写出可以为不同的 WinModem 提供一致支持的程序代码。

欲了解如何设置和操作真正的调制解调器，可参考 LDP 的“Modem HOWTO”。而 Linmodem 的设置和操作则在 LDP 的“Linmodem HOWTO”描述。因为调制解调器就是串口，之前提到的与串口有关的文档也同样适用于调制解调器。

数据采集

正如我在“数据采集模块”中所述，DAQ（数据采集）是任何程序自动化系统的基础。任何现代化的工厂或科学实验室，不管怎样都会被连接至计算机的 DAQ 描述所填满。正如前文所说，在真实世界中所发生的事情，通常可以用传感器加以量测，传感器可以将物理现象转换成电气信号。这些电气信号经过 DAQ 硬件取样之后，会被转换成软件能够读取的值。

在 Unix 或任何其他操作系统中并不存在任何标准的接口来连接 DAQ 硬件（注 11）。Comedi（control and measurement device interface）是 Linux 连接 DAQ 硬件的主要套件。你可以在 <http://www.comedi.org/> 中找到 Comedi 套件，该套件包含大量 DAQ 板的设备驱动程序。你可以在该网站的 Supported hardware 专区找到所支持 DAQ 板的完整清单。

除了为 DAQ 硬件提供驱动程序，Comedi 计划还包括 Comedilib，这是一个用户空间的链接库，可以为所有 DAQ 硬件提供跟型号或制造商无关的、统一的 API 接口。Comedilib 非常有用，因为它让你能够开发与底层硬件无关的分析软件，并可避免不受特定厂商的限制。

同样地，Kcomedilib（这是一个内核模块，可以提供类似 Comedilib 的 API）可让其他内核模块（可能执行实时的任务）存取 DAQ 硬件。

注 11： DAQ 硬件实际的形式有很多种。它可以是具 Ethernet 能力的设备、PCI 板，或者使用一些其他类型的接头。然而，供工作站使用的 DAQ 设备多半会通过标准接口（例如 ISA、PCI 或 PCMCIA）来连接。

关于 DAQ 的讨论，如果没有提到跟 DAQ 一起使用的知名商业（私有）套件，例如 LabVIEW、Matlab 和 Simulink，就不算完整。由于 Linux 在该领域非常普及，因此这三个套件已经被厂商移植到 Linux 上了。然而，请注意，已经有若干套件将开发目标定位在，为这些套件提供开放源码的替代品。例如，Scilab 和 Octave 是 Matlab 的替代品，它们的网址分别位于 <http://scilabsoft.inria.fr/> 和 <http://www.octave.org/>。

你可以在该计划的网站上找到安装和设定 Comedi 的文档以及范例。该网站还包含若干指向其他 Linux DAQ 相关网站的有用链接。至于跟封闭源码套件有关的文件，则可在相应厂商的网站上找到。

还有，某些 DAQ 硬件制造商会为它们自己的硬件以开放源码的形式或私有版权的形式提供驱动程序。当我们评估是否使用这类驱动程序时，应该将重点放在厂商未来的支持上，以免到时候受限于完全没人维护的程序代码。即使驱动程序以开放源码或自由软件的方式进行许可的，还得评估它的质量，这样当厂商决定不再支持该驱动程序的时候，还可以自己维护它。

过程控制

如同 DAQ，过程控制是任何程序自动化系统的基础。正如我在“控制模块”中所述，控制程序的方法很多，最常见的方法就是使用 PLC。最近，主流硬件（例如 PC）已经被用在程序的自动化和控制中。

将 Linux 用于控制的方法很多。首先，可以用串口或并口来驱动外部硬件，例如步进马达。这包括了前文提到的串口和并口的程序设计。就这个方法而言，并不存在任何标准的软件套件可用来连接外部所要控制的硬件，这类套件或 API 只专属于特定的应用。

你可以在既有的书籍和在线文档上找到将串口或并口用于控制的范例。如果看到的范例，讲的是执行 DOS 或 Windows 的 PC，就必须将它移植到 Linux 上才有办法加以利用。倘若要进行这样的移植工作，将会发现《Linux Device Drivers》这本书相当有用。

其次，可以使用经外围总线附接计算机的专用控制板来控制外部程序。如果是这样，需要使用该控制板专属的驱动程序。前文提到的 Comedi 套件便支持若干控制板。与此同时，控制板制造商渐渐认识到，它们的硬件有 Linux 驱动程序的需求，因此也顺势提供了这方面的支持。

最后，有一个以标准 Linux 系统取代传统 PLC 的计划正在进行中。这就是位于 <http://mat.sourceforge.net/> 的 Machine Automation Tools LinuxPLC (MAT LPLC) 计划。该计划提供了 PLC 语言解释程序、硬件驱动程序、PLC 模块同步链接库、范例模块，以及让所控制的程序可视化的 GUI 接口。LPIC 以可编程独立模块来建立 PLC，可以使用 C

语言或解释语言（例如ladder logic）进行编程（程序设计），而这些独立模块的同步运行则通过LPLC提供的某些措施（例如共享内存）来达成。因为每个模块就是一个独立的程序，所以能够轻易加入或移除控制模块。

LPLC开发团队提供的关于他们的套件如何使用和编程的信息，都可以在他们的计划网站上找到。尽管该计划仍处于初期阶段，但从开发团队所举的实际范例可看出，他们的套件已经具备实用的价值。

家庭自动化

如同DAQ以及过程控制，家庭自动化是个广泛的领域。我并不打算说明家庭自动化的基本知识或是提供必须的背景知识，因为已经有其他作者撰写过这方面的文章。如果对家庭自动化不熟悉，或是想要进一步了解这个领域，可以在Dan Hoehnen位于<http://www.homeautomationindex.com/>的Home Automation Index网站上找到大量的链接清单以及参考资料。

家庭自动化常会用到X10 Power Line Carrier（PLC，注12）协议，该技术是1970年代由一家位于苏格兰的Pico Electronics公司开发的。尽管制造商也会采用其他协议，不过X10一直都是家庭自动化中占首位的协议。

Pico后来与BSR联合成立了X10公司。直到今天，X10公司仍旧销售使用X10技术的组件。然而，就在这段时间里，最初的X10专利权已于1997年届满。因此，有许多制造商目前也提供与X10兼容的设备。X10 PLC协议让接收器和发射器之间能够在电源线上使用RF脉冲互相通信。不需要使用额外的线路，因为所有的通信都发生在房子既有的电源线上。

不像其他领域，家庭自动化在Linux方面并没有一个由中央控管的开放源码计划。取而代之的是，若干各自独立进行开发的计划。此外，内核也没有特别为家庭自动化提供驱动程序。你所需要的软件组件全都放在家庭自动化各个计划所发行的各个套件里。

以下列举家庭自动化与Linux兼容的各个开放源码计划：

MisterHouse

MisterHouse是一个完整的家庭自动化解决方案。它提供的支持包括：用户接口和X10接口软件、对各种声音进行识别的套件，以及可连接声音合成软件。

MisterHouse完全是用Perl写成的，因此可在多种操作系统上使用，包括Linux。如

注12：不要把X10 PLC跟过程控制中的PLC给搞混了。后者是实际的控制设备，与X10 PLC硬件协议无关。

果愿意接受 GPL 许可条款，可以从该计划位于 <http://misterhouse.sourceforge.net/> 的网站下载 MisterHouse 套件，该网站还备有完整的文档。

ALICE

Automation Light Interface Control Environment (ALICE) 计划。它提供的支持包括：用户接口，以及连接 X10 设备的软件。ALICE 是用 Java 写的，可以在任何适当的 JVM 上执行，包括来自 Blackdown 计划的 Linux JVM。如果愿意接受 GPL 许可条款，可以从该计划位于 <http://jhome.sourceforge.net/> 的网站下载套件，该网站还备有完整的说明文档。

HEYU!

HEYU! 只是一个命令行工具程序，可以用来控制 X10 设备。你可以到该计划位于 <http://heyu.tanj.com/heyu/> 的网站下载套件。HEYU! 采用特殊的许可条款，不过与其他的开放源码许可条款类似。你将可以在源码文件开头处找到许可条款的实际内容。

Neil Cherry 在他位于 <http://mywebpages.comcast.net/ncherry/> 的 Linux Home Automation 网站上收集了与家庭自动化和 Linux 家庭自动化计划有关的资源和链接。Neil 同时还负责维护位于 <http://linuxha.sourceforge.net/> 的计划。此计划提供了与 Linux 家庭自动化有关的链接和文档。

键盘

嵌入式系统多半不会配备键盘。虽然部分嵌入式系统会提供有限的输入接口，但是键盘通常被认为是只会在传统工作站和服务器配置中出现的奢侈品。事实上，大多数传统的嵌入式系统设计者都不认为“让嵌入式系统配备键盘”是恰当的做法。然而，近来具备 web 能力及面向消费者的嵌入式系统便附接有某种形式的键盘。

如同其他的类 Unix 系统，Linux 与用户的通信也是通过终端，以 Unix tty 为例，它就是以键盘作为输入，以控制台作为输出。这当然是将复杂的 Unix 终端 I/O 过度单纯化的说法，但是已经可以满足目前讨论的需要。因此，所有的键盘输入都会被内核当成是输入终端。尽管从用户输入的数据转换成终端的输入，其间可能涉及多个不同层的内核驱动程序，但是所有的键盘输入最后还是会馈入终端 I/O 驱动程序。

以 PC 为例，键盘的输入依次经过内核源码树中 *drivers/char* 目录里如下程序文件的处理：*pc_keyb.c*、*keyboard.c* 和 *tty_io.c*。该处理过程中的最后一个程序文件就是终端 I/O 驱动程序。基于其他架构的系统通常会使用 Input layer 机制。该机制为输入设备，例如键盘、鼠标和摇杆，明确指定了连接系统的标准方法。以 USB 键盘为例，其输入的处理

过程如下，它是从内核源码树中的 *drivers* 目录开始的：*usb/usbkbd.c*、*input/keybdev.c*、*char/keyboard.c*，然后又是 *char/tty_io.c*。

除了使用实际连接的键盘，还可以使用其他方法为终端提供输入。终端的输入还可以通过远程登录、计算机间的串行连接，以及 PDA 的手写识别软件。不论采用以上哪种方法，都可以通过编写终端 I/O 程序来存取字符输入设备。

鼠标

具有用户接口的嵌入式系统常会提供某种形式的触摸式交互接口。不论是银行的终端还是 PDA，由用户触摸屏幕产生的输入，会被当成传统工作站鼠标的输入来处理。就这点来看，许多嵌入式系统也配备有“鼠标”。事实上，提供类鼠标指示接口的嵌入式系统比提供键盘接口的嵌入式系统还多很多。

因为传统 Unix 终端并不处理鼠标的输入，所以指示设备的输入处理流程跟键盘不一样。在大多数 Linux 系统上看到的指示设备就是 */dev/mouse*，其本身通常是实际指示设备的符号链接。欲获得与指示设备动作和事件有关的信息，可以轮询和读取该设备。尽管 */dev* 目录里指示设备的文件名通常是固定的，但是从指示设备取回的数据，其格式会因设备类型而异。事实上，目前存在着许多输入格式不同的鼠标协议。请注意，鼠标使用的协议跟制造商甚至是与系统间实际连接的类型并没有直接的关系。这就是为什么 X 服务器会要求用户为鼠标设备指定一个协议的原因。另一方面，内核则提供用来管理鼠标和系统间实际连接的驱动程序。

任何涉及指示设备的程序设计都将需要对设备使用的协议有所了解。还好，已经有若干链接库和环境具有这个层次的译码实现，以及提供可用来获得指示输入和做出相应反应的 API。

显示器

明灭的 LED 和字母数字式 LCD 都是嵌入式系统传统的显示器。在嵌入式系统逐渐从许多方面侵入我们日常生活（包括服务自动化）的情况下，促使这类传统的显示方式被多样丰富的视觉接口取代。嵌入式系统在其他领域的发展，例如工厂自动化或航空电子设备，丰富的视觉接口成为标准已经好一阵子了。

正如前文所述，传统 Unix 系统提供的输出是终端形式的控制台。然而，对于目前的需求来说，这样的接口太过简单。如果没有其他支持的话，控制台只能输出文字。如果想建立图形界面的话，则需要其他更复杂的接口，其中可能包括某种形式的窗口系统。

Linux 对显示器的控制和程序设计有许多方式。虽然其中有些方式涉及内核的支持，但是主要还是依靠在用户空间上执行的程序代码，因此有利于系统的稳定，并且易于模块化。当然，为 Linux 提供图形界面的方式，还是以 X 窗口系统最常见，但是在某些环境中可能使用其他套件会比较适合。

音效

哗哗哗……，这是发射人造卫星吗，目前大多数的嵌入式系统仍旧会发出类似的声音。即使是图形非常丰富的航空电子设备和工厂自动化系统仍旧不会输出更好的声音，可能声音的分贝除外。然而，在消费性和面向服务的设备激增的情况下，音效丰富的嵌入式系统也变得越来越普遍了。

然而，Unix 的设计从未涉及音效。这些年来，出现了若干提供音效支持的方案。以 Linux 来说，主要的音效设备通常是 */dev/dsp*。其他与音效硬件有关的设备，例如 */dev/mixer* 和 */dev/sequencer*，同样具备其他音效的能力。

与在 Linux 中发展的许多其他部分相比，音效部分尚未成熟。目前有两个独立计划对声音硬件和硬件的 API 提供支持。

首先是由 Hannu Savolainen 发起的 Open Sound System (OSS)，这是个比较旧的计划。内核 2.5 版以前能够找到的声卡驱动程序，多半是根据 Hannu 提出的 OSS 架构和 API 写成的。这些年来，不仅 API 有变动，而且还经过 Alan Cox 的“整理”。然而，其中有许多部分被认为是不适合在现代的音效硬件上使用。OSS 驱动程序和 API 实际上是 Hannu 的公司 4Front Technologies 销售产品的子集，它对硬件有较广泛的支持，并且提供有较丰富的 API。你可以在 <http://www.opensound.com/pguide/> 找到与 OSS 程序设计有关的文档。

第二个音效计划是 Advanced Linux Sound Architecture (ALSA)。ALSA 的主要目标是提供一个完全模块化的音效驱动程序套件，以及在 API 和管理架构这两方面提供优于 OSS 的环境。就硬件支持而言，ALSA 计划支持 OSS 不支持的硬件。因为 ALSA 计划以及它所有的组件都是以开放源码的形式发行，所以你可以在该计划位于 <http://www.alsa-project.org/> 的网站获得所有文件和源码。

有人预测，ALSA 计划将会在内核里某些地方取代 OSS 的内容。从 Linux 2.5 开始，ALSA 的确被整合进内核，尽管目前还暂时使用 OSS 驱动程序。如同 Linux 对其他领域的支持，对音效的支持因目标板架构而异。越主流的架构获得的支持也越好。

打印机

如同许多主流的外围设备一样，嵌入式系统通常不支持打印机。然而，也有例外。支持打印功能的嵌入式 Web 服务器，就是一个嵌入式系统需要操作系统支持打印机的好例子。传统的嵌入式系统开发者通常会认为“嵌入式 Web 服务器”这个说法有矛盾，但是提供这类服务的设备越来越常见，而且涉及的开发方式与那些应用较受局限的嵌入式设备类似。

传统的 Unix 对打印机的支持，与许多其他操作系统提供的支持相比过于老旧。不幸的是，Linux 对打印机的支持主要是根据其他的 Unix 打印系统。已经出现若干计划将目标定位在为 Linux 和一般的 Unix 提供容易使用和模块化的打印服务。

为了解如何打印文件，以及在 Linux 中如何实现对打印机的支持，让我们来看看一般文件打印的步骤，从用户的打印请求开始，到打印机实际打印出文件为止：

1. 用户提交所要打印的文件。可以在命令行上或是通过图形应用程序的菜单来完成此事；在 Linux 中，这两种做法有些细微的差异。通常，Unix 程序的打印输出，采用的是 PostScript (PS) 的格式，这种格式的输出会被送往打印机。市场上并非所有打印机都具备处理 PS 格式的能力，PS 格式的输出必须转换成实际打印机可以处理的格式。如果有必要，转换的工作会在稍后的阶段完成。
2. 文件本身及用户指定的打印选项，会被存入打印机专属的单个队列中。该队列可能位于本地（如果打印机直接附接在系统上）也可能位于远端（如果打印机附接在位于网络的服务器上）。无论队列位于何处，用户根本不必知道它的处理过程。
3. 由一个 spooling 系统负责管理打印队列，并在打印机空闲的时候，根据文件在队列中的先后次序进行打印。将文件从 PS 格式转换成打印机可以识别的实际格式，就是在这个阶段完成的。转换工作通过一组过滤器进行，一个过滤器只负责一种格式的转换。其中最重要的是 PS-to-printer 过滤器，该过滤器适合各种类型的打印机。

根据使用的打印管理软件不同，以上的步骤可能会有些细微的差异。目前 Linux 有五种打印管理套件可用：LPD、PDQ、LPRng、CUPS 和 PPR。LPD 是个传统的套件，可以在大多数的发行套件里发现它的踪迹。其他套件渐有进展，慢慢会取代 LPD 的地位。然而，不论你使用哪种套件，将 PS 格式转换成打印机格式的工作，通常会交由 GhostScript（注 13）来完成，GhostScript 是个非常重要的套件，可用来检查和操作 PS 格式的文档。一旦完成转换，输出最后会被馈入实际的打印机设备，不管它是一台并口打印机还是一台 USB 打印机。

注 13： GhostScript 需要使用大量的内存空间，可能不适合使用在小型的嵌入式 Linux 系统上。

别忘了，以上所有的工作都会为用户空间完成。内核驱动程序只会在过滤后的输出被馈入实际打印机时介入。

如果想让嵌入式系统支持打印机，建议研读与传统 Unix 或 Linux 系统管理有关的书籍，以便获得 Unix 打印机管理方面的知识。Welsh、Dalheimer 与 Kaufman 合著的《Running Linux》(O'Reilly) 对 Linux 如何使用 LPD 完成打印机的设置有很好的描述。与 Linux 打印功能有关的最新信息，可以在这个议题的主要资源网站 <http://www.linuxprinting.org/> 找找看。你还可以在这个网站上找到“Printing HOWTO”，该文档包含了各种打印管理套件的操作说明以及相关链接。

存储设备

所有嵌入式系统的启动都至少需要使用某种形式的永久性存储设备，即使是在引导过程的最初期阶段。大多数系统，包括嵌入式系统，仍会继续使用同一个存储设备来进行它的其余操作（执行程序代码或是存取数据）。然而，与传统的嵌入式软件相比，在嵌入式系统中使用 Linux 对存储硬件的需求，不论就规模或结构来看都比较大。

规模需求已经在第一章讨论过了，而典型的存储设备配置也在第二章说明过了。我将会在第 7 章和第 8 章进一步探讨实际的结构。这一节，让我们来看看 Linux 支持的永久性存储设备。尤其是，我们将讨论 Linux 对这些设备的支持程度以及典型用法。

MTD

在 Linux 的术语中，memory technology device（存储技术设备，MTD）涵盖了所有存储设备，例如常见的 ROM、RAM、flash 以及 M-Systems 的 DiskOnChip (DOC)。正如 Michael Barr 在《Programming Embedded Systems in C and C++》(O'Reilly) 中提到的，这类设备的能力、特性和限制，各不相同。因此，为了在自己的系统中编程和使用 MTD 设备，嵌入式系统开发者传统上会使用该类设备专属的工具和方法。

为了尽可能避免针对不同的技术使用不同的工具，以及为不同的技术提供共同的能力，Linux 内核纳入了 MTD 子系统。它提供了一致且统一的接口，让底层的 MTD 芯片驱动程序无缝地与称为“用户模块”的较高层接口组合在一起，如图 3-1 所示。这些“用户模块”不应该跟内核模块或任何用户空间的软件搞混。“MTD 用户模块”指的是内核中的软件组件，它们会借着提供可识别的接口和抽象层，让内核的较高层或用户空间能够存取底层 MTD 芯片驱动程序。

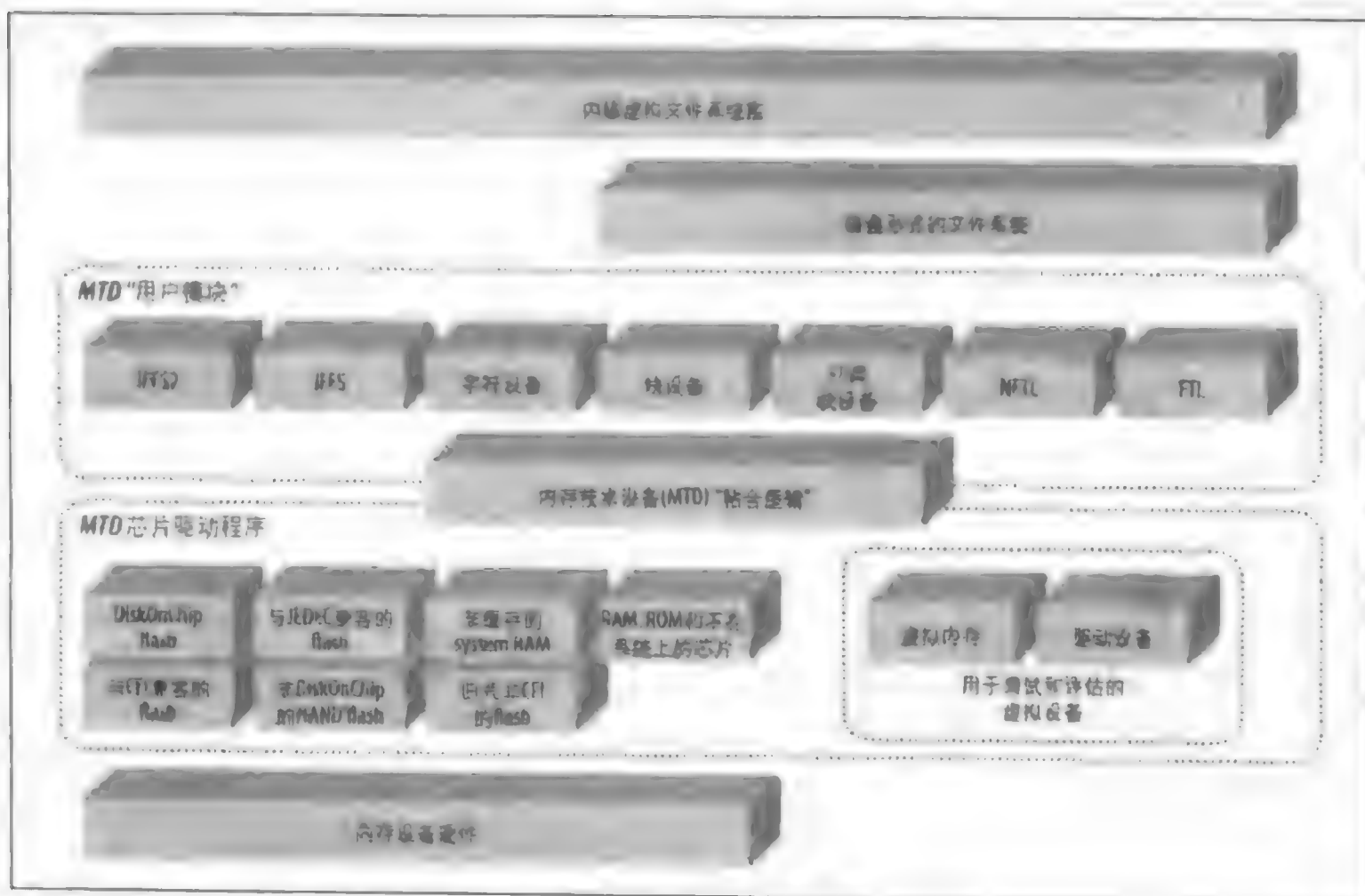


图 3-1: MTD 子系统

MTD 芯片驱动程序必须向 MTD 子系统注册，方法是通过 `mtd_info` 结构给 `add_mtd_device()` 函数提供一组缺省的回调函数及属性。MTD 驱动程序必须提供回调函数，让 MTD 子系统能够调用它来执行删除、读出、写入和同步等操作。以下列出目前已经可以使用的 MTD 芯片驱动程序：

DiskOnChip (DOC)

这是 M-Systems 公司 DOC 技术的驱动程序。目前，Linux 支持 DOC 1000、DOC 2000 和 DOC Millennium。

Common Flash Interface (CFI)

CFI 是一个由 Intel、AMD 和其他 flash 制造商共同开发的规格。所有与 CFI 兼容的 flash 组件都会将它们的配置和参数直接存放在芯片上。因此，它们的侦测软件接口、配置及使用能够标准化。内核中内含侦测和支持 CFI 芯片的程序代码。

因为 CFI 规格允许不同的芯片使用不同的命令，所以目前内核的支持包括了两个不同芯片系列（Intel/Sharp 和 AMD/Fujitsu）所实现的两组指令集。

JEDEC

JEDEC Solid State Technology Association (<http://www.jedec.org/>) 是一个为 flash 芯片定义标准的组织，它还负责为此类设备分发识别编号。尽管 CFI 的出现使得

JEDEC flash 标准被淘汰出局，不过目前仍旧存在与 JEDEC 兼容的芯片。MTD 子系统支持此类设备的侦测和配置设定。

非 DOC 的 NAND flash

NAND flash 是 M-Systems 公司的 DOC 设备最普遍的规格。然而，市场上还有其他类型的 NAND flash 芯片。MTD 子系统对一些这种设备使用非 DOC 的驱动程序。该驱动程序支持设备的完整列表，可参考内核源码树中的 *include/linux/mtd/nand-ids.h* 文件。

旧式非 CFI 的 flash

有些 flash 芯片既不兼容 CFI 也不兼容 JEDEC。因此，MTD 子系统会根据制造商的规格为此类设备提供驱动程序。这种设备被支持的形式包括：非 CFI 但与 AMD 兼容的 flash 芯片、pre-CFI 的 Sharp 芯片，以及非 CFI 的 JEDEC 设备。然而，不要忘了，此类设备的更新并不如较常用的设备（例如 DOC 或 CFI 设备）来得频繁。

RAM、ROM 和不在系统上的芯片

MTD 子系统提供存取一般 RAM、ROM 芯片的驱动程序，并将它映射到系统的物理寻址空间，就像 MTD 设备一样。因为此类芯片与系统的连接有些可能会使用插座或其他可让你移除芯片的类似接头，所以 MTD 子系统还提供一个驱动程序，可于从系统上移出设备时，用来保存 MTD 设备节点的注册顺序。

非缓存的 RAM

如果有任何系统 RAM 是 CPU 无法缓存的，在系统正常操作期间你可以将此内存当成 MTD 设备来使用。当然，当你关掉系统的电源时，存放在这类媒体上的信息将会消失。

用于测试和评估的虚拟设备

当你要为目标板上的设备加入或测试 MTD 的支持时，有时你可能会想要进行与芯片驱动程序无关的用户模块的操作测试。为此目的，MTD 子系统提供了两种可用来模拟实际 MTD 硬件的 MTD 驱动程序：其中一个驱动程序模拟 MTD 设备的方式是使用系统的虚拟寻址空间提供的内存，另一个驱动程序模拟 MTD 设备的方式是使用一般的块设备。

因为不同的 MTD 设备在物理地址上并不会映射到相同的位置，所以 MTD 子系统需要使用经过裁剪的 mapping 驱动程序（注 14）以便检查及管理系统上的 MTD 设备。有些系统和开发板知道 MTD 设备的配置，因此内核包含了此类系统专属的映射驱动程序。内核还包含了一般驱动程序，可用来存取系统上没有专属映射驱动程序的 CFI flash 芯片。

注 14：映射驱动程序是一种特殊的 MTD 驱动程序，它的主要任务是给 MTD 芯片驱动程序提供 MTD 设备在系统中的物理地址和一组存取这些设备的函数。

如果系统的存储设备没有适当的映射驱动程序可用,可能需要以既有的映射驱动程序为蓝本自己建立一个新的。你可以在内核源码树中的 *drivers/mtd/maps/* 目录里找到既有的映射驱动程序。

如同其他的内核设备驱动程序,一个 MTD 芯片驱动程序可以管理同一种设备的多个实体。例如,系统上有两个与 CFI 兼容的 AMD flash 芯片,它们可以被 CFI 驱动程序的单一实体当成是两个不同的 MTD 设备来管理,不过这取决于设置(注 15)。为了让 MTD 设备里的存储空间易于定制,MTD 子系统还允许把存储设备划分成多个分区。与硬盘分区非常像的是,每个 MTD 分区可以当成是独立的 MTD 设备来存取,在同一个设备上的不同分区可以使用完全不同的格式来存储数据。实际上,正如我们在第二章所见,存储设备通常会被划分成许多分区,每个分区的用途各不相同。

在我们为系统的存储设备设好 MTD 芯片驱动程序的配置之后,每个 MTD 设备上的存储空间将可被一个 MTD 用户模块管理。用户模块将会对它管理的 MTD 设备使用某种存储格式,以及(正如之前所说的)提供较高层内核组件能识别的接口和抽象层。切记,MTD 用户模块并无法与所有的 MTD 驱动程序互通。事实上,某些 MTD 用户模块可能因为技术上或甚至是法律上的限制而无法使用某些 MTD 驱动程序。例如,在写作本书时,让 JFFS2 用户模块使用 NAND flash 设备的开发正在进行中。直到最近,却演变成 JFFS2 用户模块无法使用任何形式的 NAND flash(包括 DOC 设备),因为 JFFS2 无法处理 NAND flash 芯片的某些特性。然而,修正此状态的工作已在进行中,当你看到此处的内容时,JFFS2 可能已经可以使用 NAND 设备了。以下列出现有的 MTD 用户模块及其特性:

JFFS2

JFFS2 是 JFFS 的后继者,由 Red Hat 重新改写而成。JFFS2 的全名为 Journalling Flash File System Version 2(闪存日志型文件系统第 2 版),其功能就是管理在 MTD 设备上实现的日志型文件系统。与其他的存储设备存储方案相比,JFFS2 并不准备提供让传统文件系统也可以使用此类设备的转换层。它只会直接在 MTD 设备上实现日志结构的文件系统。JFFS2 会在安装的时候,扫描 MTD 设备的日志内容,并在 RAM 中重新建立文件系统结构本身。

除了提供具有断电可靠性的日志结构文件系统,JFFS2 还会在它管理的 MTD 设备上实现“耗损平衡”和“数据压缩”等特性。

断电可靠性对嵌入式系统来说很重要,因为它们可能会在任何时间失去电力。然后系统必须按部就班地重新引导,并且在不需要外力介入的情况下恢复文件系统的内

注 15: 放在系统总线上的同一种芯片,通常会被安排成就像一个大型的芯片一样。

容。如果 Linux 或甚至是 Windows 工作站突然失去电力，重新引导的时候，可能必须等待系统检查文件系统的完整性，而且系统可能会提示你手动进行一些检查。通常，嵌入式系统无法接受这种情况；它可以按部就班地恢复文件系统而且不受断电的影响。然而，请注意，它并不保证能够恢复遭中断的文件系统操作。例如，一个应用程序调用 `write()` 用新数据覆盖旧数据，可能会在旧数据被部分覆盖且新数据尚未完全提交的时候突然失去电力。于是这两组数据都会丢失。系统应该在启动的时候检查这种失败。

之所以需要“耗损平衡”这个功能，是因为 flash 设备中每个块的抹除操作有次数上的限制，通常是 100 000 次，但不同的制造商之间可能会有差异。一旦块的抹除次数达到这个上限，制造商便无法保证该块的正确运行。为避免过度使用某些块让设备的寿命因而缩短，JFFS2 实现了一个算法，可确保 flash 设备上所有区块具有一致的使用率，因此可以平衡每个区块的损耗程度。

因为 flash 硬件的价格通常比速度较慢的 RAM 硬件还贵，所以存放在 flash 设备上的数据最好先经过压缩以节省空间，并在使用数据之前将它解压缩到 RAM 上。这正好是 JFFS2 要做的事。因此，在 JFFS2 中不可能使用 `eXecute In Place`（就地执行，XIP，注 16）。

JFFS2 已经被广泛地用作各种 MTD 设备的文件系统选项。例如，Familiar 计划 (<http://familiar.handhelds.org>) 就是以 JFFS2 来管理 Compaq iPAQ 里的 flash 设备。

正如前文所说，尽管 JFFS2 目前无法在 NAND 设备（包括 DOC 设备）上使用，不过现在正在构建中，可能在你阅读此处内容时已经可以了。其间，JFFS2 可用在其他类型的 MTD 设备上，有时甚至可以用在 CompactFlash 设备上，CompactFlash 设备实际运行时就像连接在系统 IDE 接口上的 IDE 硬盘一样。

NFTL

NAND Flash Translation Layer（NAND Flash 转换层，NFTL）在 NAND flash 芯片上实现了一个虚拟块设备。如图 3-1 所示，然后必须使用一个磁盘形式的文件系统，例如 FAT 或 `ext2`，通过受 NFTL 管理的 MTD 设备存取数据。

切记，M-Systems 公司持有 NFTL 实现的算法的专利权，因此这些算法只能应用在经过许可的 DOC 设备。尽管 NFTL 本身具有断电可靠性，不过你还是在 NFTL 上面使用日志型文件系统，这样，系统的存储设备才不会受断电的影响。例如，一个在 NFTL 上使用 `ext2` 的嵌入式系统死机了，与一般 Linux 工作站非常像的是，它必须在重新启动的时候进行文件系统的完整性检查。

注 16： XIP 是一种无须将程序代码复制到 RAM 直接从 ROM 执行的能力。

JFFS

Journalling Flash File System (闪存设备日志型文件系统, JFFS) 最初由瑞典的 Axis Communications AB 开发, 其目的是作为嵌入式系统免受宕机和断电危害的文件系统。然而据说已经可以用于 NAND 设备上的 JFFS (在你阅读此处内容时 JFFS2 可能已经具有此特性) 将会大量地被 JFFS2 取代。

FTL

Flash Translation Layer (Flash 转换层, FTL) 在 NOR flash 芯片上实现了一个虚拟块设备。如同 NFTL, 还必须使用一个“真正的”文件系统来管理经 FTL 操作的设备。

FTL 也有专利权的问题。在美国, 它只能用在 PCMCIA 硬件上。为了替代在 NOR flash 芯片上使用 FTL 这个方案, 可能会想要直接使用 JFFS2, 因为它没有任何专利权的困扰, 所以是个比较好的解决方案。

字符设备

这个用户模块可以对 MTD 设备进行类似字符设备的存取操作。使用它, 每个 MTD 设备可以直接当成字符设备 (以 Unix 的观点) 来操作。它最适合应用在 MTD 设备的初始设置中。正如我们将在第七章所见, 这是一个让此类字符设备读写的数据生效, 必须进行的特殊方法。例如, 将数据写入字符设备之前, 通常必须先进行抹除的操作。

高速缓存块设备

这个用户模块为 MTD 设备提供了一个块设备接口。它让一般工作站和服务器的文件系统能够在这些设备上使用。然而, 只有少数需要 JFFS2 功能的商业嵌入式系统会这么做, 这种模块最适合使用在不必先删除分区内容就可以将数据写入 flash 分区的应用。它还可以用来设置以只读方式本地安装文件系统的系统。

这种模块之所以被称为“高速缓存”块设备用户模块, 是因为它的运行是通过 RAM 中的高速缓存块, 首先根据需要修改它们, 接着抹除适当的 MTD 设备块, 然后重新写入遭到修改的块。当然, 此处无法提供断电可靠度。

只读块设备

除了没有实现 RAM 高速缓存功能, 只读块设备用户模块的能力如同高速缓存块设备。因此文件系统中所有内容都是只读的。

正如你所见, MTD 子系统的组成既丰富又复杂。尽管 MTD 用户模块与 MTD 芯片驱动程序之间正确的对应规则让它的使用变得极为复杂, 不过它具备相当的弹性并且能够有效地对存储设备提供一致的存取接口。你可以在 Memory Technology Device Subsystem 计划位于 <http://www.linux-mtd.infradead.org/> 的网站上找到与实现 MTD 用户模块及

MTD 芯片驱动程序的 API 有关的文档。该网站还提供 MTD 邮件论坛以及 Vipin Malik 撰写的极为详尽的“MTD-JFFS-HOWTO”文档。

我们将会在第七章继续讨论 MTD 子系统，以及详述在嵌入式系统中使用 MTD 设备要如何设置以及设定配置。

ATA-ATAPI (IDE)

AT Attachment (ATA, 注 17) 是 1986 年由三家公司: Imprimis、Western Digital 和 Compaq 开发的。最初只有 Compaq 在使用 ATA，但最后当 Conner Peripherals 开始通过零售店提供它的 IDE 磁盘驱动器时，它已经变得相当普及了。1994 年 ATA 成为 ANSI 标准。之后，为了提升其传输速度及扩充能力，该标准陆续发展出了几个不同的版本。此外，CD-ROM 制造商在 Western Digital 和 Oak Technology 的协助之下发展出了 ATA Packet Interface (ATA 包接口, ATAPI)。ATAPI 让系统能够使用类似 SCSI 的命令包通过 ATA 接口存取 CD-ROM 和磁带机。目前负责发展和维护 ATA 和 ATAPI 的单位包括 ANSI、NCITS 和 T13。

尽管只有少部分的传统嵌入式系统会需要永久性存储媒体提供像 IDE 硬盘那样多的存储空间，但是许多嵌入式系统通常都会使用与 ATA 兼容的 flash 设备，例如极普遍的 CompactFlash。与“存储设备”中探讨的 flash 设备相比，CompactFlash 的存储空间只能使用 ATA 接口进行存取。因此，从软件的观点来看，甚至从硬件的观点来看，它简直就是一个小型的 IDE 磁盘驱动器。请注意，CompactFlash 卡还可以通过 CompactFlash-to-PCMCIA 适配卡来存取。我们将会在第七章深入探讨 Linux 使用 CompactFlash 设备的方法。在此之前，不要忘了，并非所有的 CompactFlash 设备都适合在嵌入式系统中使用。尤其是，某些 CompactFlash 设备无法容忍突然失去电力，这可能会造成永久性的损害。

在嵌入式系统中设置 IDE 和大多数其他类型磁盘的方式，通常会像在工作站或服务器上一样。一般来说，磁盘上会存放操作系统引导加载程序、根文件系统，有时还包括 swap 分区。然而，与大多数工作站和服务器相比，并不是所有的嵌入式系统监控程序和引导加载程序都具备 ATA 的能力。事实上，正如我们将在第九章看到的，大部分的引导加载程序都不具备 ATA/IDE 的能力。如果想在自己的系统上使用 IDE 磁盘，但是该系统的 flash 中并没有具备 ATA 能力的监控程序或引导加载程序，则必须把内核和引导监控程序一同放在 flash 或 ROM 中，这样才能在系统启动的时候存取到它。然后，必须设定引导监控程序，让它在启动的时候使用该内核，以便存取 IDE 磁盘。在这种情况下，仍旧可以在 IDE 设备上设定根文件系统以及 swap 分区。

注 17: 尽管它通常被称为“IDE”(全名为 Integrated Drive Electronics)，不过“ATA”才是该接口的真正名称。

Linux 对 ATA 接口的支持相当广泛并且非常成熟。ATA 子系统（位于内核源码树中的 *drivers/ide* 目录里）包括许多芯片的支持以及缺陷修正。这项支持横跨许多架构。此外，内核还支持 PCMCIA IDE 设备，并且提供可以用在 ATAPI 设备上的 SCSI-emulation 驱动程序。后者结合 SCSI 驱动程序之后，可以用来控制仍不存在 ATAPI 原生驱动程序的 ATAPI 设备。尽管从 2.5 版内核开始就不再需要此驱动程序，不过该功能对配备 CD-RW 设备的工作站用户最为有用，因为在 Linux 中可用来操作这些设备的工具，其底层的硬件必须是 SCSI。

由于考虑到 ATA/IDE 支持的重要性，在内核邮件论坛上与其有关的大多数修改和更新公告都会直接整合进内核。相对而言，其他子系统的维护者则会通过子系统的计划网站独立提供最新的版本，这些维护者会不时送出一个或（通常是）一组补丁给 Linux，让 Linux 更新内核（包括稳定版）。然而，ATA/IDE 的相关工具，*hdparm* 和 *fdisk*，的维护工作并不属于内核的范畴，主要是因为它们是用户工具而不是内核正常操作必须的组件。*hdparm* 会使用内核中 ATA/IDE 驱动程序支持的 `ioctl()` 命令来获得和设定 IDE 硬盘参数。*fdisk* 可用来检查和修改磁盘分区。如果曾经在工作站上安装过 Linux，对 *fdisk* 或许已经很熟悉。请注意，该工具程序不限于在 IDE 硬盘上使用，它也可以在 SCSI 磁盘上使用。

想了解与 Linux 的 ATA/IDE 性能有关的信息，位于 <http://www.linux-ide.org/> 的 Linux ATA Development Project 网站是主要的起点。该网站除了提供与 ATA 相关的用户工具，还提供关于 ATA 的许多链接。位于内核源码树中 *Documentation* 目录里的 *ide.txt* 文档也很重要，它提到了内核对 IDE 设备的支持以及如何设定内核才能正确存取此类设备。

你还可以找到一些不局限在 Linux 的 ATA/IDE 接口的在线资源和出版物。Robert Bruce Thompson 与 Barbara Fritchman Thompson 合著的《PC Hardware in a Nutshell》(O'Reilly) 用了一章的篇幅论述 IDE 和 SCSI 硬盘接口，书中对它们的优缺点作了比较。尽管该书的讨论集中在高层的议题上，不过它对 ATA/IDE 世界有很好的介绍，并且可作为你选择硬盘接口的参考。至于更深入的探讨，可能会想看看《Enhanced IDE FAQ》（可以从 <http://www.faqs.org/> 获得），该文件提供的提示和技巧是从 *comp.sys.ibm.pc.hardware.storage* 新闻群组收集来的。最后，如果想清楚了解 ATA 接口，可以向 ANSI 购买相关的标准文件。然而，在你这么做之前，务必检查内核源码树中相关的部分，因为它们通常会包含难得的信息。

SCSI

正如“*I/O*”中提到的，使用 SCSI 存储设备的嵌入式 Linux 系统并不多见。在嵌入式 Linux 系统中设置和设定这些设备的方法跟在服务器中非常像。因此，或许可以参考任何适当

的系统管理书籍或在线文档提供的指示和建议。当然，在“I/O”节中提到的文件和资源仍旧管用。至于《PC Hardware in a Nutshell》(O'Reilly)一书则简述了 SCSI 存储设备，并且比较了它与 ATA/IDE 的优缺点。

通用网络

有越来越多的嵌入式系统被连接到通用网络上。尽管就许多方面来说，这些设备受到的限制都会比其他的计算机系统还多，不过它们所提供的服务通常都可以在许多现代的服务器中找到。好在 Linux 本身很适合在通用网络上使用，因为它通常会在主流的服务器中使用。

接下来的讨论涵盖了嵌入式系统中最常见的网络硬件。Linux 对网络硬件的支持比我要讨论的范围还要广，不过其中有许多网络接口通常不会在嵌入式系统上使用，因此我会忽略它们。此外，因为这些网络接口在别处已经有广泛的说明，所以我将会把讨论限制在与嵌入式系统有关的话题上，并且指引你到其他来源获得进一步的信息。

我们将会在第十章进一步讨论网络服务相关议题。

Ethernet

Ethernet 最初由 Xerox 位于 Palo Alto, California 的 PARC 研究中心开发，它目前是使用最普遍、文档最佳、价格最便宜的网络类型。它的速度能够跟上竞争者，这十年来以几何级数成长。考虑到 Ethernet 的普遍性以及嵌入式系统对网络需求的增加，许多嵌入式开发板和产品系统在出货时都会提供 Ethernet 硬件。

Linux 支持许多种 10 和 100 Megabit 的 Ethernet 设备和芯片。它还支持若干千兆 Ethernet 设备。内核建立配置菜单或许是你检查内核是否支持特定硬件的最佳起点，因为它包含了最新的驱动程序清单（注 18）。LDP 的“Ethernet HOWTO”还包含了 Linux 所支持硬件的清单，以及关于在 Linux 上使用 Ethernet 的许多信息。最后，写过若干 Linux Ethernet 驱动程序的 Donald Becker 在 <http://www.scyld.com/network/> 维护了一个网站，目的在于提供与 Linux 的网络驱动程序有关的信息。

有若干资源探讨到 Ethernet 的使用和内幕。Charles Spurgeon 著作的《Ethernet: The Definitive Guide》(O'Reilly)便是一个好的起点。Charles 还在 <http://www.host.ots.utexas.edu/ethernet/> 维护了一个提供 Ethernet 资源的网站。其中“Ethernet FAQ”的内容是从

注 18： 你或许还会想要使用此清单作为自己设计硬件的基础，正如我在前文中的建议。

comp.dcom.lans.ethernet 新闻群组收集来的。如果需要为自己的硬件撰写 Ethernet 驱动程序，将会发现《Linux Device Drivers》这本书很有用。

IrDA

Infrared Data Association（红外线数据协会，IrDA）于1993年由50家公司成立，目的在于建立和推动便宜的红外线数据互连标准。IrDA的第一份规格于1994年发布，并由协会继续维护和发展，因此这份规格以协会的名字为名。今日，可以在许多消费性设备中，包括PDA、蜂窝电话、打印机以及数字相机等等，找到IrDA的硬件和软件。与其他的无线方案例如Bluetooth相比，IrDA价格低廉。因此得到了广泛采用。

IrDA规格里的协议主要分成两种：必要和选用。一个设备至少要实现必要协议才能够正确地与其他的IrDA设备互通。必要协议包括物理信号（physical signaling, IrPHY）层、链路存取协议（link access protocol, IrLAP）层以及链路管理协议（link management protocol, IrLMP）。最后一个协议还包含信息存取服务（Information Access Service, IAS），该服务具备查找的能力。

IrDA设备能够在1米的距离内以最高4 Mbps的传输速率交换数据。与其他无线技术不同的是，IrDA需要参与通信的设备互相指向对方。这么做显然可以增加安全性，因为在整个联机期间，参与通信的IrDA设备必须互相指向对方（注19）。

Linux支持所有必要的IrDA协议以及许多选用的协议。图3-2展示了Linux的IrDA子系统的架构。

IrPHY是实际的红外线设备，数据便是通过它来传送的。它通常会被放在所属设备的边缘。以PDA为例，它通常会被放在设备顶端，这样，当用户将自己的IrDA指向另一个用户的PDA或任何其他具IrDA的设备时，他还可以看到PDA的屏幕。

IrDA标准是根据传输速率来对IrPHY设备进行分类的。目前共分成三类：最高可达115.2 Kbps的串行红外线（serial infrared, SIR）、最高可达1.152 Mbps的中速红外线（medium speed infrared, MIR）以及4.0 Mbps的快速红外线（fast infrared, FIR）。未来，16 Mbps的极快速红外线（very fast infrared, VFIR）应该可以成为标准的一部分。

Linux内核为SIR和FIR设备提供以下驱动程序：

IrTTY

IrTTY用来支持与16550-UART兼容的IrDA端口。此驱动程序会使用内核的串行驱动程序并能提供最高可达115200 bps的传输速率。

注19：任何“入侵者”都必须直接面对参与通信的用户。

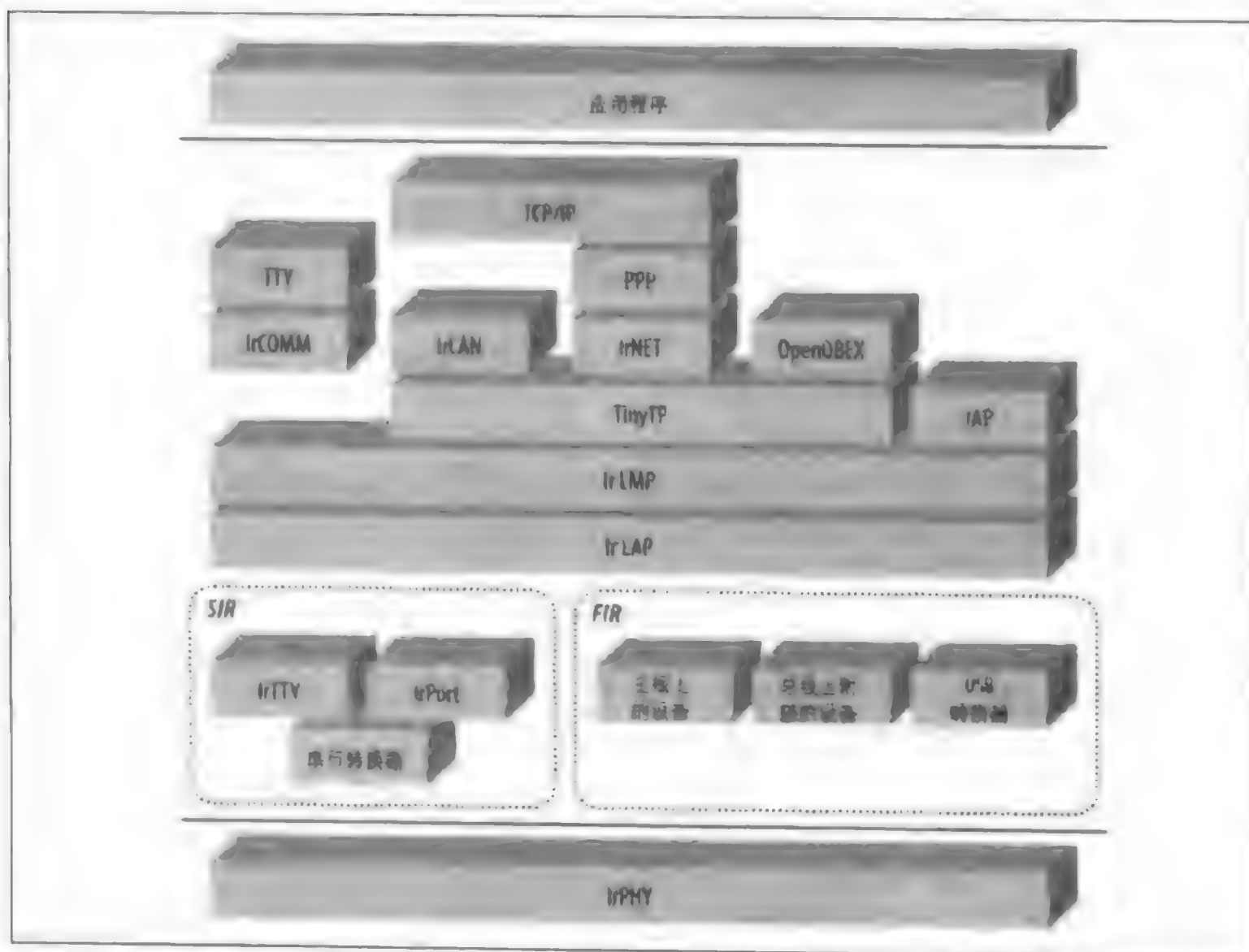


图 3-2: LinuxIrDA 子系统架构

IrPORT

IrPORT 是个半双工（half-duplex）串口驱动程序，主要用来取代 IrTTY。

串行转换器（serial dongle）

这是一个能够附接在系统串口上的 IrDA 转换器，可以为没有内建 IrDA 端口的系统提供 IrDA 的支持。内核建立配置菜单包含了 Linux 支持的串行转换器的完整清单。

主机板上和总线附接设备

内核支持主机板上和总线附接的 IrDA 设备中可以找到的若干芯片。内核建立配置菜单包含了 Linux 支持芯片的完整清单。

USB 转换器

如同串行转换器，USB 转换器提供的是可移除的 IrDA 接口。不同的是，USB 转换器具有 FIR 等级的传输率。

IrDA 堆栈运行在设备驱动程序之上，用于在 Linux 中提供 IrDA 的功能。此堆栈的大部

分组件是根据 IrDA 标准实现的，其中有若干组件的实现并非标准的一部分。以下是实现在内核中的 IrDA 堆栈的各个层：

IrLAP

IrLAP 是 IrDA 规格的链路存取协议层。用来提供和维护 IrDA 设备间的可靠连接。除了一般的面向连接协议，Linux 还支持以 Ultra 协议进行无连接的通信。

IrLMP

IrLMP 是 IrDA 协议的链路管理协议层。用来提供和管理 IrLAP 上的多重链接。

TinyTP

小型传输协议 (Tiny Transport Protocol, TinyTP) 实现 IrLMP 链接上的流量控制。

IAP

Linux 的信息存取协议 (Information Access Protocol, IAP) 与 IrDA 的信息存取服务 (Information Access Service, IAS) 等效。如同 IAS 一样，IAP 用来提供服务查找的能力。

IrCOMM

IrCOMM 是仿真层，可为经由一般串口和并口通信的既有应用程序提供 IrDA 连接的能力。因为这类功能通过 TTY 来存取，所以应用程序会使用内核的 TTY 层来存取 IrCOMM。请注意，IrCOMM 不需要经由 TinyTP。

IrLAN

IrDA 规格中的 IrLAN 使 IrDA 设备间能够使用类似局域网的连接。从上层协议的观点来看，IrLAN 就像是一个网络设备。例如，在 IrDA 连接上使用 IrLAN 可以建立 TCP/IP 网络。

IrNET

IrNET 也可以让 IrDA 设备间使用类似局域网的连接。与 IrLAN 不同的是，IrNET 并未实现成功能完整的网络设备，它只实现了最基本的部分，不过 PPP 与它并用可以成为一个完整的网络设备。然而，请注意，IrNET 并非官方 IrDA 标准的一部分。Microsoft 首次推出 IrNET 的时候，它是 Windows 2000 IrDA 堆栈的一部分，用来取代 IrCOMM 和 IrLA。Linux 对 IrNET 的实现概念是根据 Microsoft 的 IrNET，因此可以彼此互通。

OpenOBEX

IrDA 规格中的 IrOBEX，是一个类似 HTTP 的服务，可用来交换对象。OpenOBEX 是 Linux 的 IrOBEX 实现。组成它的用户空间链接库可以在 <http://sourceforge.net/projects/openobex/> 中找到。

除了IrDA堆栈,还需要用户空间工具来操作Linux的IrDA能力。这些工具是IrDA Utils套件的一部分。该套件以及与IrDA有关的许多其他资源都可以在位于<http://irda.sourceforge.net/>的Linux-IrDA Project网站中找到。

欲进一步了解Linux的IrDA堆栈以及相关的工具和计划,可以参考LDP的《Infrared HOWTO》文件。此外, Linux-IrDA计划的主要贡献者Jean Tourrilhes,在http://www.hpl.hp.com/personal/Jean_Tourrilhes/维护了若干非常值得参考的Linux-IrDA网页。与其他标准不同的是,IrDA标准的所有文件都可以从协会位于<http://www.irda.org/>的网站直接下载。

IEEE 802.11 (无线网络)

IEEE 802委员会于1990年设置了802.11工作小组。该工作小组于1997年发表了第一个802.11标准,并由同一个小组继续维护和更新。此标准支持计算机间以2.4 GHz (802.11b)和5 GHz (802.11a)的频率进行无线通信。现在,802.11被广泛采用,并且受到主流的支持,支持的程度就像无线版的Ethernet。

尽管许多传统的嵌入式设备都配备某种形式的无线技术,不过近来支持802.11的嵌入式系统大多是面向用户的设备,例如PDA。将此类设备连上802.11网络的方法跟膝上计算机或工作站差不多。因此,稍后所提到的参考数据,只须花点功夫就能应用在具802.11能力的嵌入式设备上。

Linux对802.11b硬件有广泛的支持。欲了解Linux支持哪些802.11硬件以及相关的驱动程序和工具,建议参考Jean Tourrilhes所写的《Linux Wireless LAN HOWTO》(可以在http://www.hpl.hp.com/personal/Jean_Tourrilhes/找到该文件)。内核包括对主机板上(on-board)或非PCMCIA总线附接(non-PCMCIA bus-attached)设备的支持,而且可以在内核建立配置菜单上加以选择。另一方面,对PCMCIA总线附接的802.11网卡的支持则包含在“PCMCIA”中提到的David Hinds的PCMCIA套件中。

因为大多数802.11设备的操作跟Ethernet设备类似,所以内核不需要对它们提供任何额外的支持。而且,一旦适当的设备驱动程序完成加载和初始化的动作后,通常应用在Ethernet设备上的工具,多半也可以应用在802.11设备之上。然而,若干802.11的特性需要专门的工具来处理,例如识别码和密钥的设定、信号强度的监控以及链接的质量。这些工具都可以到Jean的网站上的Wireless Tools for Linux专区获得。

除了Jean的网站,LDP的“Wireless HOWTO”还提供有Linux使用无线设备的若干背景信息。如果想要大量使用802.11设备,可能会想要看一下Matthew Gast著作的《802.11 Wireless Networks: The Definitive Guide》(O'Reilly)。这本书对802.11技术以及Linux

如何使用 802.11 设备有完整的探讨。你还可以从 IEEE 获得实际标准的副本。写作本书时，还可以从 IEEE 的网站自由下载这些副本的 PDF 文档。Jean 的网站提供有适当的链接，不过你应该注意与 IEEE 的网站有关的链接是否仍旧有效。

Bluetooth

1994 年，Ericsson 在 Intel 的协助之下，推出 Bluetooth（蓝牙）标准。Bluetooth SIG 最初的成员包括 Ericsson、IBM、Intel、Nokia 和 Toshiba。现在，Bluetooth SIG 的成员已超过 1900 家公司。目前，有广泛的设备，例如 PDA 和蜂窝电话，具备 Bluetooth 的能力。

Bluetooth 运行在 2.4 GHz 频带上，并且使用扩频跳频的技术，可用来在相同的 piconet 上提供设备的无线连接性（注 20）。有人称 piconet 为“cable 的替代品”，也有人称它为“无线 USB”。基本上，它让设备之间能够进行无缝的无线通信。因此，Bluetooth 设备不需要经过任何配置设定就可以成为 piconet 的一部分。更确切地说，每个设备会自动侦测其他设备并公布自己的服务，这样 piconet 中的其他设备随后便可以使用这些服务。

Linux 具有若干 Bluetooth 堆栈。主要的堆栈有四个：BlueZ、OpenBT、Affix 和 BlueDrekar。BlueZ 最初由 Qualcomm 编写，不过它现在是个开放源码计划，如果愿意接受 GPL 的许可条款，可以从该计划位于 <http://bluez.sourceforge.net/> 的网站获得该堆栈。接下来的讨论会将重点放在 BlueZ，因为它就是主流内核源码树中所包含的 Bluetooth 堆栈。

OpenBT 由 Axis Communications AB 发展和维护。它就放在该计划位于 <http://developer.axis.com/software/bluetooth/> 的网站上。与 BlueZ 相比，OpenBT 的文档和源码注释比较好。但是 OpenBT 被构建成串行抽象层（它的存取是经由 `/dev/ttyBT0`、`/dev/ttyBT1` 等设备文件），而 BlueZ 则被构建成网络协议（通过 `AF_BLUETOOTH` 这个 socket 类型对它进行存取），就许多方面而言，这样对 Bluetooth 比较适合，因为它本身就是一个协议。

Affix 由 Nokia 发展和维护。它就放在位于 <http://affix.sourceforge.net/> 的 SourceForge 网站上。相关的 user-space 工具和内核补丁也可以到该网站下载。如同 BlueZ，它也被构建成一个网络协议——通过 `AF_AFFIX` 这个 socket 类型对它进行存取。

最后，BlueDrekar 堆栈由 IBM 发展和散布。它就放在该计划位于 <http://www.alphaworks.ibm.com/tech/bluedrekar/> 的网站上。尽管 BlueDrekar 可供任何人自由下载，但是它并不是一个开放源码计划，因此我将不会对它做进一步的讨论。

注 20：piconet 是个由 Bluetooth 设备所组成的无线网络。因为 Bluetooth 设备可以同时隶属多个 piconet，所以 piconet 可以重叠。

图 3-3 展示了 BlueZ 堆栈的架构。如果熟悉 Bluetooth，便会注意到 BlueZ 并不支持 Telephony Control protocol Specification Binary (TCS-bin) 或 OBEX (注 21)。尽管 Linux 支持 IrDA OBEX，但是在我写作本书时，既有的 Linux 实现 OpenOBEX 无法与 OpenBT 一起运行，只能使用 BlueZ 的初级功能。这是因为 OpenBT 并未实现 OBEX 而且 BlueZ 对 OBEX 的支持仅处于早期阶段。然而，OpenOBEX 可以跟 Affix 一起使用，因为 Affix 实现了 OBE。

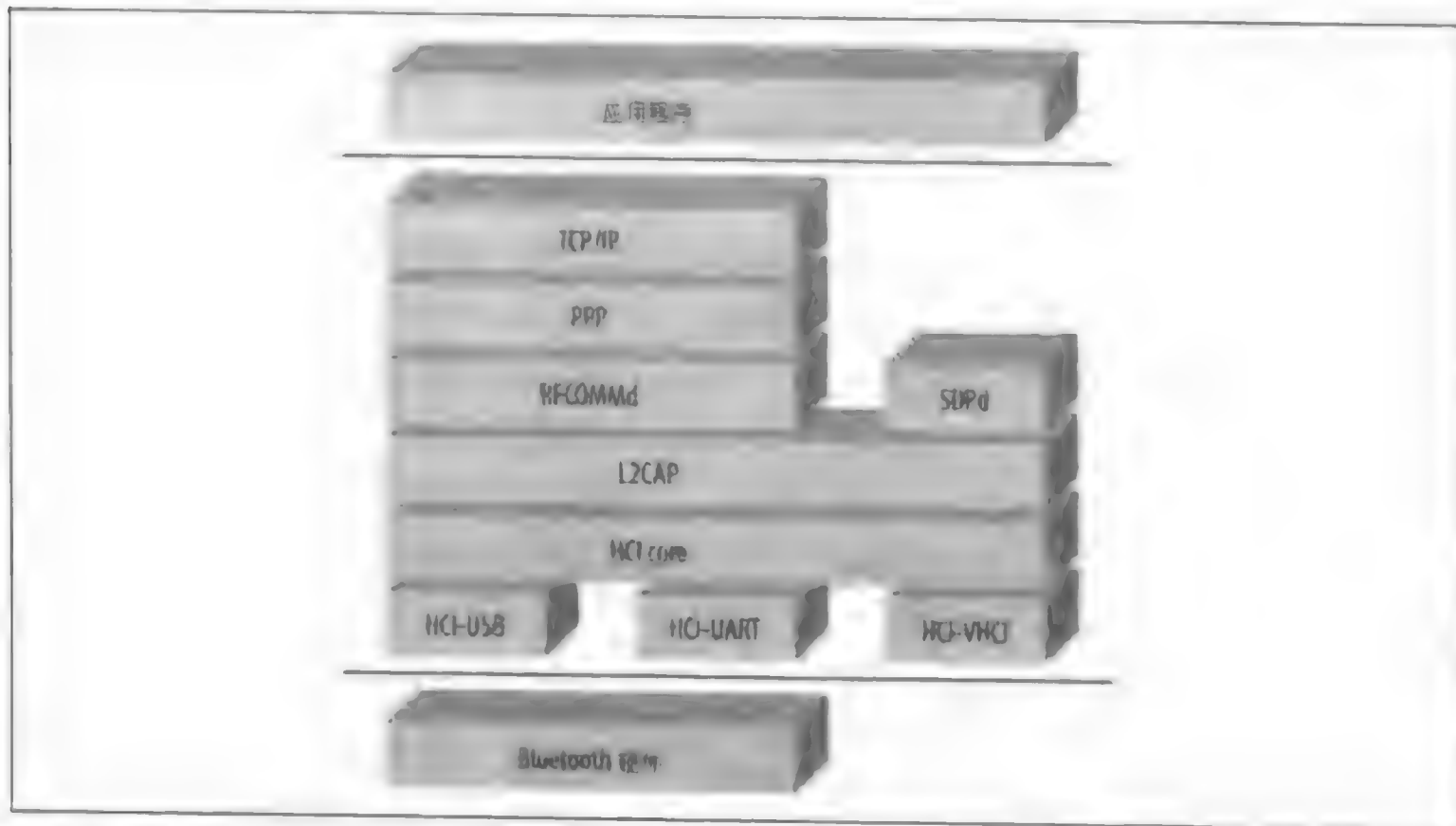


图 3-3: Linux 的 BlueZ Bluetooth 子系统架构

Host Controller Interface (HCI) 是 Bluetooth 堆栈中最低的一个层，负责连接实际的 Bluetooth 硬件。BlueZ 目前支持以下类型的 HCI 适配卡：

HCI-USB

附接到 USB 上的 Bluetooth 设备。不要与内核建立配置菜单的 USB support 次级菜单中找到的“USB Bluetooth”支持给搞混了，后者需要的是 OpenBT，而不是 BlueZ。

HCI-UART

附接到串行接口上的 Bluetooth 设备。

注 21: 这就是 IrDA 标准中的 OBEX 规格。为了避免发明新的协议，Bluetooth 标准直接将 IrDA OBEX 规格拿来使用在 HTTP-like 二进制代码交换服务的实现上。

HCI-VHCI

VHCI代表虚拟HCI。因此，VHCI的功能就像虚拟Bluetooth设备，可用于测试和开发。

HCI硬件设备驱动程序接着会通过HCI core跟协议堆栈的较高层互通。BlueZ包括下列协议层：

L2CAP

Logical Link Control and Adaptation Protocol(逻辑链路控制及适配协议,L2CAP)负责链路复用、包分割与重组以及服务质量。

RFCOMMd

Bluetooth标准的RFCOMM协议可为Bluetooth设备提供串行通信的能力。BlueZ将RFCOMM实现成监控程序RFCOMMd，使用pseudo-TTY进行通信。如果接着在RFCOMMd上使用PPP，则可以让Bluetooth设备间使用TCP/IP进行通信。

SDPd

Service Discovery Protocol(服务查找协议,SDP)让每个设备能够查找其他具备Bluetooth能力的设备提供的服务，以及公告每个设备提供的服务。在BlueZ中，SDP被实现成监控程序SDPd。

除了协议堆栈，将需要从BlueZ计划的网站获得user-space工具。除了RFCOMMd和SDPd，还包括*l2ping*（用于L2CAP pinging）和*hcidump*（用于HCI包分析）。

下面是可供参考的相关资源：

- 欲进一步了解BlueZ堆栈的操作，可参考该计划网站上的“Linux BlueZ HOWTO”。
- 与OpenBT以及它的使用有关的信息，可在该计划为于<http://sourceforge.net/projects/openbt/>的SourceForge工作空间获得。
- Delbert Matlock在<http://www.selfstudycerts.com/extra/linux-bluetooth.html>维护了一份Linux Bluetooth资源清单，如果打算以Linux来使用Bluetooth，可能会发现这份清单很有用。
- Prentice Hall出版了两本受欢迎的Bluetooth书籍：另一本是Brent Miller与Chatschik Bisdikian合著的《Bluetooth Revealed: The Insider's Guide to an Open Specification for Global Wireless Communications》，另一本是Jennifer Bray与Charles Sturman合著的《Bluetooth: Connect Without Cables》。

你可能还有兴趣成为Bluetooth SIG的会员。Bluetooth SIG的网站位于<http://www.>

bluetooth.org/。你可以从位于<http://www.bluetooth.com/>的正式Bluetooth网站获得正式的Bluetooth标准。

工业级网络

如同其他的计算机应用，工业控制及自动化也逐渐依赖计算机网络。然而，通用网络或联机解决方案，例如一般的Ethernet或Token Ring，不适合在工业应用所处的恶劣环境中使用。例如，常见的Ethernet，若在大多数的工业环境中使用，很容易受到电磁干扰（Electromagnetic Interference, EMI）以及无线频率干扰（Radio Frequency Interference, RFI）的影响。

因此，随着时间的流逝，发展了不少专用的、工业级的网络解决方案。除了更适合在工业环境中应用，这些工业网络（一般所谓的现场总线）的贡献包括缩减线路、增加模块性、提供诊断能力、可进行自我配置设定，以及简化企业信息系统的设置。

接下来，我将会说明Linux支持的工业网络，以及简述Linux为什么支持或不支持的其他工业网络。如果是现场总线的新手，可能会想要看一看Rob Hulsebos的现场总线页面（位于<http://ourworld-top.cs.com/rahulsebos/>）。该网站收集了大量各种现场总线系统的相关链接和参考资料。

CAN

Controller Area Network（控制器局域网络，CAN）不仅是最常见的现场总线，可能也是最普遍的网络形式之一。CAN是1986年由Robert Bosch GmbH为汽车工业提出的串行总线系统，此后被沿用在许多其他的工业中。CAN发展初期有Mercedes-Benz工程师的贡献和Intel提供的第一颗CAN芯片82526。而今，每年都销售超过一亿的新CAN设备。应用范围从较高级的汽车（例如Mercedes）到工厂自动化网络。

CAN的规格包括硬件接口以及通讯机制。它是个多主控者的串行网络协议，具备错误侦测能力，信号的辨识通过内容（而不是接收节点或是传送节点）来完成。CAN标准（由CAN in Automation (CiA) 小组负责管理和推动）被提交为1993年所出版的ISO 11898标准。

因为CAN是个底层协议，如同Ethernet一样，有许多高层协议被提出以弥补其不足。这类协议包括J1939、DeviceNet、Smart Distributed System (SDS) 以及CANopen。极受汽车工业欢迎的J1939协议由Society of Automotive Engineers（汽车技师公会，SAE）提出并负责维护。DeviceNet是另一个受欢迎的以CAN为基础的高层协议，该协议由Open DeviceNet Vendor Association (ODVA) 负责管理。SDS由Honeywell提出，并

由 Honeywell 负责推动和管理。CANopen 由维护 CAN 标准的同一个团体 CiA 提出和负责管理。SDS 并不如 DeviceNet 和 J1939 流行，因为它一直都不是标准，而 J1939、DeviceNet 和 CANopen 则是标准。

尽管内核并未正式支持 CAN，不过有许多开放源码计划为 Linux 提供 CAN 硬件的支持。以下列出其中最重要的部分：

Linux CAN-bus Driver Project (Linux CAN 总线驱动程序计划)

这是一个主要的开放源码 CAN 支持计划。此计划提供的内核模块支持许多以 Intel 82527 和 Philips sjal000 为基础的 CAN 电路板。此计划位于 <http://home.wanadoo.nl/arnaud/> 的网站备有文档、指南和链接（CAN 相关网站）等专区。

Alessandro Rubini 的 Ocan 驱动程序

这是一个以 Intel 82587 为基础的 CAN 电路板的驱动程序。它的维护者是《Linux Device Drivers》(O'Reilly) 的作者之一。如果接受 GPL 的许可条款，则可以从 <http://www.linux.it/~rubini/software/#ocan> 获得该驱动程序，那里还备有非常好的说明文档。

can4linux

can4linux 以往由 Linux Lab Project 负责维护。现在可以从 http://www.port.de/engl/canprod/sw_linux.html 获得该套件。此套件包括以 Philips 82c200 为基础的电路板的驱动程序以及应用范例。

CanFestival

CanFestival 为 ADLINK PCI 7841 电路板在 Linux 中提供了 CAN 和 CANopen 的能力。为该电路开发的软件可以在标准的 Linux 系统上应用，并且可以在 RTLinux 骨架的实时系统上使用。此套件及其文件可以从 <http://canfestival.sourceforge.net/> 获得。

ss5136dn Linux 驱动程序

此套件为 SST 5136-DN 系列的 CAN bus/DeviceNet 接口板在 Linux 中提供了 CAN 和 DeviceNet 的能力。此套件（包括详细的文件以及用户空间链接库）可以从 <http://www.spectra-one.com/dn5136man.html> 获得。

欲进一步了解 CAN、CAN 相关硬件以及 CANopen，可查阅 CiA 位于 <http://www.can-cia.org/> 的网站。CiA 会在线提供其规范文档。SAE 只让订阅者从其位于 <http://www.sae.org/products/j1939.htm> 的网站获得 J1939 标准的文档。你可以从 ODVA 位于 <http://www.odva.org/> 的网站获得关于 DeviceNet 的信息。你可以付费从 ODVA 获得印刷版的 DeviceNet 规格书，此费用包括了复制的钱以及开发 DeviceNet 产品终身免权利金。

的许可。如果对 SDS 感兴趣，可以在 Honeywell 位于 <http://content.honeywell.com/sensing/prodinfo/sds/> 的网站找到进一步的信息，包括规格书。

ARCnet

Attached Resource Computer NETwork (ARCnet) 是由 Datapoint Corporation 于 1977 年提出的通用网络，与 Ethernet 非常像。而今，ARCnet 几乎不再用于办公室局域网中，不过它仍然是受欢迎的工业现场总线。ARCnet 现在是 ANSI 标准并由 ARCnet Trade Association (ATA) 负责管理和推动。

ARCnet 是个以令牌为基础的网络，它可以采用星状架构或是总线架构。任何一张 ARCnet 网络适配卡只能兼容于其中一种架构。除了价格低廉，与标准的办公室网络相比，ARCnet 还有许多优点，包括性能稳定、自动重设配置、具有多主控者能力以及具备抗噪声能力。此外，ARCnet 能够保证包的安全送达，以及传送失败时保证通知对方。

Linux 内核支持 ARCnet 已经有相当一段时间了。因为各种 ARCnet 网卡的程序设计接口几乎一样，所以不需要使用各式各样的设备驱动程序。内核包含两种标准 ARCnet 芯片组 (COM90xx 和 COM20020) 的驱动程序。除了驱动程序，内核还包含三种可用在 ARCnet 硬件上的协议。第一个也是最常见的协议，符合 RFC1201 的规范，用来在 ARCnet 网络上传送 IP 流量。例如，当系统被设定成使用 RFC1201 协议时，内核自己的 TCP/IP 堆栈用来在 ARCnet 硬件上提供 TCP/IP 网络连接。第二个协议符合 RFC1051 的规范，这个规范已经被前述的 RFC1201 取代。此协议可让系统与旧网络交互。最后，内核还提供 Ethernet 封包协议，此协议让 ARCnet 网络能够传送 Ethernet 包。

关于 Linux ARCnet 驱动程序的信息可以从 ARCnet for Linux 计划位于 <http://www.worldvisions.ca/~openwarr/arcnet/> 的网站获得。该网站还提供了对 Linux 使用 ARCnet 有广泛探讨的“Linux-ARCnet HOWTO”文件。这份 HOWTO 文件包含了许多 ARCnet NIC 的跳线设定信息以及卡片的图解。它还会告诉你如何架设 ARCnet 网络。这份 HOWTO 文件的文字副本也包含在内核源码树中的 *Documentation* 目录里。

ATA 位于 <http://www.arcnet.com/> 的网站包含更多关于 ARCnet 的信息，其中包括 ANSI 标准和其他手册的订购单。

Modbus

Modbus 协议由 Modicon 于 1978 年提出，它以主从方式在控制器和传感器之间使用 RS232 传送控制数据。并购 Modicon 之后，Schneider Electric 拥有 Modbus 商标权并继续带领此协议及其后继协议的开发。

Modbus 制定的传输结构与底下的物理层无关。Modbus 使用两种格式 (ASCII 和 RTU) 来传送信息。第一种格式会将每个字节传送成两个 ASCII 字符, 而第二种格式则会将每个字节传送成两个 4 位的十六进制字符。Modbus 通常会在 RS232、RS422 或 RS485 之类的串行接口上实现。除了 Modbus, Schneider 还制定了 Modbus TCP/IP 协议, 该协议可以使用 TCP/IP 和 Ethernet 来传送 Modbus 信息。

目前有两个开放源码计划为 Linux 提供 Modbus 的能力:

jModbus

此计划的目的在于提供 Modbus RTU、Modbus ASCII 和 Modbus TCP/IP 的 Java 实现。此计划采用的是 BSD 形式的许可条款, 可以在 <http://jmodbus.sourceforge.net/> 获得套件、说明文件以及范例。

MAT LinuxPLC

我曾在“程序控制”中提到这个自动化计划。MAT 计划的 CVS 库中现在包含有实现 Modbus RTU 和 Modbus TCP/IP 的程序代码。尽管源码有注释, 不过它并没有提供其他的文件。

欲进一步了解 Modbus 以及阅读 Modbus 规格书请访问 <http://www.modbus.org/>。

其他工业网络

当然, Linux 不支持的其他工业网络也有很多。例如, Linux 尚不支持 ControlNet、Seriplex、AS-Interface 和 Sercos。而且 Linux 可能支持其他形式的现场总线, 不过 Linux 要支持它们, 还需要多费一番功夫。以下所列的就是这类现场总线:

Interbus

此驱动程序可让 2.0 和 2.2 版的内核支持 Phoenix Contact Interbus 板子。此驱动程序附带说明文档和范例, 可以从 <http://www.santel.lu/projects/wallace/interbus.html> 获得。

LonWorks

此驱动程序可让 2.2 版的内核支持 EasyLon 接口。此驱动程序仅供评估之用并附带若干文件和范例, 可以从 http://www.gesyttec.de/englisch/support/linux_readme.htm 获得。

同时, Linux 内核中还有 Applicom 卡的驱动程序。尽管驱动程序的作者主要的目的是将它应用在 Profibus, 不过 Applicom 卡可以处理许多协议。使用时, Applicom 卡会被当作 `/dev` 目录里的字符设备。

此外, Hilscher Gmbh. 为它的 CIF 板提供了一个设备驱动程序, 以及可用来开发现场总线与应用程序无关的用户层框架, 相关信息参见 Hilscher 位于 <http://www.hilscher.com/> 的网站。套件中包含的设备驱动程序, 可以从用户空间使用一致的框架 API 来进行存取。这让我们能够开发出与底层现场总线技术无关的控制应用。尽管此套件目前只有 Hilscher 的硬件驱动程序, Hilscher 使用的方法以及所提供的框架有助于 Linux 在未来为工业网络技术提供广泛且一致的支持。

系统监控

硬件和软件常常出问题, 有时甚至完全停止运行。尽管通过小心的设计和运行测试可以降低问题的发生概率, 但这有时是无法避免的。嵌入式系统设计者的任务就是做好准备, 提供恢复的方法。通常, 故障检测以及恢复会使用系统监控硬件和软件 (例如 watchdog) 来进行。

Linux 支持两种系统监控措施: 看门狗定时器、硬件健康状态监控器。看门狗定时器同时具有硬件和软件的实现, 而健康状态监控器总是需要适当的硬件。看门狗定时器会根据重初始化周期来判断系统是否需要重新引导。如果系统停机, 定时器最后会因为计时超时而重新引导。而硬件健康状态监控器则会提供系统硬件状态的信息。接着可根据此信息采取适当的动作, 例如发出通知或解决实际的硬件问题, 如过热或电压不稳定。

内核包括许多看门狗定时器的驱动程序。你可以在内核建立配置菜单中的 Watchdog Cards 次级菜单里找到内核支持的看门狗设备的完整清单。此清单包括看门狗定时器外围卡的驱动程序、软件看门狗, 以及某些 CPU (例如 MachZ 和 SuperH) 里可以找到的看门狗定时器驱动程序。尽管你可能会想要使用软件看门狗来省掉硬件看门狗的花费, 但是不要忘了在某些情况下软件看门狗可能无法让系统重新引导。在 Linux 中, 看门狗定时器可被视为 `/dev/watchdog`, 必须周期性地对它执行写入动作以避免系统重新引导。这个更新任务传统上是由 watchdog 监控程序 (可至 <ftp://metalab.unc.edu/pub/linux/system/daemons/watchdog/> 获得) 来进行的。然而, 在实际的嵌入式系统中, 可能会想让主应用程序而不是使用看门狗监控程序来进行更新, 因为后者可能无法确切地知道主应用程序是否已经停止运行。

除了 Linux 内核提供的软件看门狗, RTAI 还提供可设定策略的复杂的软件看门狗。RTAI 看门狗的主要目的是保护系统以免受到 RTAI 应用程序中设计错误的影响。因此, 行为不端的任务不会造成系统停机。侦测到引起问题的任务之后, RTAI 看门狗可以对它进行若干补救措施, 包括暂停它、杀掉它以及延长它的周期。RTAI 看门狗以及相关的文件附带在主流的 RTAI 发行套件中。

最后，位于 <http://www2.lm-sensors.nu/~lm78/> 的 Hardware Monitoring by lm_sensors 计划网站为 Linux 支持了不少的硬件监控设备。此计划的网站包含了所支持设备的完整清单，以及关于软件安装和操作的大量文档。lm_sensors 套件（可从该计划的网站获得）包含设备驱动程序以及连接驱动程序的用户层工具程序。这些工具程序包括 *sensord*，这是一个监控程序，它会登录感应值，当警报条件发生时它会以 ALERT 这个 syslog 等级向系统发出警报。该网站还提供了外部计划及 lm_sensors 相关资源的链接。



与上流软件开发者非常类似，嵌入式系统开发者也需要用到编译器、链接器、解释程序、集成开发环境以及诸如此类的其他开发工具。然而，嵌入式开发者的工具有所不同，因为他们用来执行应用程序的平台与用来建立应用程序的平台并不相同。因此，这些工具常被称为跨平台开发工具或简称交叉开发工具。

本章探讨的是交叉开发工具的设置、配置和使用。首先，我将会探讨如何使用实际的项目工作空间。接着，我将会探讨GNU跨平台开发工具链、C链接库的替代方案、Java、Perl、Python、Ada、其他程序语言、集成开发环境，以及终端仿真程序。

实际项目工作空间的使用

在你为目标板开发及定制软件的过程中，将会需要在一个综合的、容易使用的目录结构里，组织各种软件包和项目组件。表4-1展示了我建议的目录安排方式。请自行修改此目录结构以符合自己的需要与需求。当你要决定各组件的存放位置时，务必找出最直观的安排方式。同时，自己的程序代码务必与从网络下载的包分开存放。这么做可以把因源码所有权和许可状态所造成的任何疑义减到最少。

表4-1：建议的项目目录安排方式

目录	内容
<i>bootldr</i>	目标板的引导加载程序
<i>build-tools</i>	建立跨平台开发工具链需要用到的包和目录
<i>debug</i>	调试工具以及所有相关包
<i>doc</i>	项目将会用到的所有文档
<i>images</i>	准备使用在目标板上的引导加载程序和内核的二进制映像，以及根文件系统
<i>kernel</i>	你将在目标板上进行评估的各个内核版本

表 4-1: 建议的项目目录安排方式 (续)

目录	内容
<i>rootfs</i>	目标板的内核在运行时看到的根文件系统
<i>sysapps</i>	目标板需要用到的系统应用程序
<i>tmp</i>	进行实验时或临时文件会用到此目录
<i>tools</i>	跨平台开发工具链以及 C 链接库全都放在此目录

当然，以上这些目录还会包含许多子目录。本书之后的章节会陆续加上这些目录。

至于项目工作空间要放在何处，可以自己决定，不过我强烈建议不要放在系统目录，如 */usr* 或 */usr/local*。我建议你最好放在自己的主目录，或是 */home* 目录底下可让开发小组所有成员共享的目录。如果真的想要放在系统目录里，或许可以考虑 */opt* 这个目录。以第一章提到的嵌入式控制系统为例，我将此系统的工作空间安排在自己的主目录里：

```
$ ls -l ~/control-project
total 4
drwxr-xr-x 13 karim karim 1024 Mar 28 22:38 control-module
drwxr-xr-x 13 karim karim 1024 Mar 28 22:38 daq-module
drwxr-xr-x 13 karim karim 1024 Mar 28 22:38 sysmgnt-module
drwxr-xr-x 13 karim karim 1024 Mar 28 22:38 user-interface
```

由于此系统的各个组件全都在不同的目标板上运行，所以在我的主目录中，这些组件在 *control-project* 目录底下都各自有专属的目录。每个组件都各自拥有如上所述的项目工作空间。例如，*daq-module* 是数据采集模块的工作空间：

```
$ ls -l ~/control-project/daq-module
total 11
drwxr-xr-x 2 karim karim 1024 Mar 28 22:38 bootldr
drwxr-xr-x 2 karim karim 1024 Mar 28 22:38 build-tools
drwxr-xr-x 2 karim karim 1024 Mar 28 22:38 debug
drwxr-xr-x 2 karim karim 1024 Mar 28 22:38 doc
drwxr-xr-x 2 karim karim 1024 Mar 28 22:38 images
drwxr-xr-x 2 karim karim 1024 Mar 28 22:38 kernel
drwxr-xr-x 2 karim karim 1024 Mar 28 22:38 project
drwxr-xr-x 2 karim karim 1024 Mar 28 22:38 rootfs
drwxr-xr-x 2 karim karim 1024 Mar 28 22:38 sysapps
drwxr-xr-x 2 karim karim 1024 Mar 28 22:38 tmp
drwxr-xr-x 2 karim karim 1024 Mar 28 22:38 tools
```

因为你将要建立与使用的某些工具程序，可能会用到这些目录的路径，此时若能设计一个简短的命令脚本，用来设定适当的环境变量，或许有帮助。下面这个称为 *develdaq* 的命令脚本，便是用来设定 DAQ 模块的环境变量：

```
export PROJECT=daq-module
export PRJROOT=/home/karim/control-project/${PROJECT}
cd $PRJROOT
```

除了设定环境变量，这个命令脚本还会将你移往包含此项目的目录。如果不想马上移往该项目所属的目录，可以移除命令脚本中的 `cd` 命令。要在目前的 shell 中执行此命令脚本，以便让环境变量立即生效，可键入（注 1）：

```
$ . develdaq
```

往后的说明将会视为环境变量 `PROJECT` 和 `PRJROOT` 已经存在。

警告： 因为你将为目标板建立的包，有许多都是发行包已经安装在工作站上的，所以务必将这两种软件划分清楚。要有清楚的划分，我强烈建议，当你以 `root` 的身份登录工作站时，切勿执行书中所提到的任何指令，除非我有明确指示可以这么做。此外，这将可避免，系统安装的原生 GNU 工具链（最重要的是，大多数应用程序依赖的 C 链接库）遭到任何可能的破坏。因此，切勿使用 `root` 的身份，请改用不具有特权的一般用户账号登录工作站。

GNU 跨平台开发工具链

为了对任何目标板进行应用程序的交叉开发，我们需要将各种二进制工具程序集成进工具链，其中包括如 `ld`、`gas`、`ar`、C 编译器（`gcc`）以及 C 链接库（`glibc`）。往后各章的内容都必须依赖我们在此处收集的跨平台开发工具链。

你可以从 FSF 的 FTP 网站 <ftp://ftp.gnu.org/gnu/> 或任何其他镜像网站下载 GNU 工具链的各个组件：`binutils` 包位于 `binutils` 目录，`gcc` 包位于 `gcc` 目录，而 `glibc` 包则与 `glibc-linuxthreads` 包一起放在 `glibc` 目录。如果使用的 `glibc` 版本比 2.2 还旧，还必须下载 `glibc-crypt` 包，该包同样位于 `glibc` 目录。链接库的这个部分之所以要分开发行，是因为美国密码学输出法规定，美国境外的计算机不管是从 FSF 的网站，还是从美国境内的任何其他网站下载该包，都是不合法的行为。然而，自从 2.2 版开始，`glibc-crypt` 已经整合进 `glibc` 套件中了，因此不再需要额外下载此套件（注 2）。根据前文建议的项目目录安排方式，这些包应该下载到 `${PRJROOT}/build-tools` 目录。

注 1： 提到任何 shell 命令时，本书都会假定所使用的是 `sh` 或 `bash`，因为它们是最常用到的 shell。如果使用的是另一种 shell，应该依实际的情况修改命令。

注 2： `glibc` 开发者邮件论坛有一封信提到既能将 `glibc-crypt` 整合进 `glibc` 套件又能符合美国输出法规定的办法，参见 <http://sources.redhat.com/ml/libc-alpha/2000-02/msg00104.html>。这封信与接踵而至的讨论有提到 BXA 这个缩写。BXA 指的是美国商务部（U.S. Department of Commerce）的工业与安全局（Bureau of Industry and Security），这个单位之所以会简称为 BXA，是因为它的前身是出口管理局（Bureau of Export Administration）。

请注意，GNU 工具链支持第三章提到的所有目标板。

GNU 工具链入门

设定及建立适当的 GNU 工具链是件复杂棘手的工作，需要对不同软件包的依存关系以及它们各自的角色功能有相当的了解。这是必备的知识，因为 GNU 工具链中各个组件的开发与发行完全是各自独立的。

组件的版本

建立工具链的第一步就是选择我们将会使用的组件版本。这包括 binutils 的版本、gcc 的版本以及 glibc 的版本。因为这些包的维护与发行完全是各自独立的。当与其他套件的不同版本组合在一起时，并非一个套件的所有版本都能够顺利完成建立的工作。你可以试着使用每个套件的最新版本，但这种组合并不保证一定管用。

想要选用适当的版本，必须试着找到适合主机和目标板的组合。当然，或许会发现比较简单的做法是，四处打听看看周围是否已经有人对同类型的设置测试过某种版本的组合，并且该组合能够正常运行。然而，如果打听不到这样的版本组合，到头来你可能还需要为自己的设置测试可用的版本组合。如果真是这样，一开始请使用每个套件最新出来的稳定版本，如果无法建立的话，再依次换成较旧的版本。

警告： 有时候，编号最高的版本可能测试的时间太短，还称不上“稳定”。例如，在 glibc 2.3 发表之后 glibc 2.3.1 发表之前，或许较好的选择还是继续使用 glibc 2.2.5。

举个例子，在写作本书时，binutils 的最新本版是 2.13.2.1，gcc 的最新本版是 3.2.1，而 glibc 的最新本版是 2.3.1。binutils 通常都能够建立成功不需要做任何改变。因此，假定 gcc 3.2.1 建立失败，尽管我已经正确提供了所有的配置标志。此时，我会回头使用 gcc 3.2。如果还是失败，我会尝试 gcc 3.1.1，直到建立成功为止。同样地，glibc 2.3.1 也会建立失败。此时，我会回头使用 glibc 2.3，如果需要的话，我还会尝试 glibc 2.2.5。

然而，必须了解的是，无法像这样毫无限制地回头尝试以前的版本，因为最新出来的套件版本会认为其他套件理应提供某些能力。因此，如果其他套件的最新版本建立失败，已经建立成功的套件也应该回头使用较旧的版本。承前例，如果我必须回头使用 glibc 2.1.3，我可能需要回头使用 gcc 2.95.3 和 binutils 2.10，尽管最新版的 gcc 和 binutils 或许已经编译成功。

你可能还需要对某些版本进行修补的动作，让它们能够顺利完成编译，供目标板使用。

第三章为每种处理器架构所提供的网站和邮件论坛,就是你寻找此类补丁文件和包版本建议的最佳处所。另一处可以找到补丁文件的地方,就是Debian源码包:每个包都会包含该包支持的每种架构所需要的补丁文件。

表 4-2 提供的是已知可用的版本组合。该表针对主机 / 目标板的组合,提供了 binutils、gcc 和 glibc 可以互相搭配的版本组合。该表的最后一列用来指示相应的工具是否需要进行修补。

表 4-2: 已知可用的包版本组合

主机	目标板	内核	binutils	gcc	glibc	进行修补
i386	PPC		2.10.1	2.95.3	2.2.1	是 ^a
i386	PPC		2.11.2	2.95.3	2.2.1	是 ^a
PPC	i386		2.10.1	2.95.3	2.2.3	否
PPC	i386		2.13.2.1	3.2.1	2.3.1	否
i386	ARM	2.4.1-rmk1	2.10.1	2.95.3	2.1.3	是 ^b
PPC	ARM		2.10.1	2.95.3	2.2.3	是 ^b
i386	MIPS		2.8.1	egcs-1.1.2	2.0.6	是 ^c
i386	SuperH		2.11.2	3.0.1	2.2.4	是 ^d
Sparc (Solaris)	PPC	2.4.0	2.10.1	2.95.2	2.1.3	否

- a. 关于如何为 MPC860 PPC 建立工具链的进一步信息,请参考 Karim Yaghmour 的说明 (<http://www.embeddedtux.org/pipermail/etux/2003-June/000103.html>)。
- b. 关于如何修改让 gcc 得以建立成功的进一步信息,请参考 AlephOne 写的《Guide to ARMLinux for Developers》(<http://www.aleph1.co.uk/armlinux/book/book1.html>) 这本书,“The GNU toolchain”这一章,“Building the Toolchain”这一节,“The -Dinhibit_libc hack”这一小节。
- c. 关于如何修补的进一步信息,请参考 Ralf Bachle 所写的“Linux/MIP SHOWTO”(<http://howto.linux-mips.org/>)。
- d. 关于如何修补的进一步信息,请参考 Bill Gatliff 所写的“Running Linux on the Sega Dreamcast”(<http://www.linuxdevices.com/articles/AT7466555948.html>)。

表 4-2 呈现的某些组合是跨平台开发工具链设置的一部分。在最初说明提到的内核版本的组合,表中也会呈现该内核版本。然而,事实上内核版本对建立工具链并不重要。任何已知能够有效运行在目标板上的新近内核版本(2.2.x 或 2.4.x)都可以用来建立工具链。然而,强烈建议使用跟目标板一样的内核版本,以避免日后发生冲突。我将会在第 5 章探讨如何选用内核。

尽管表 4-2 并未提及，不过我们的工具链将会需要 glibc 的附加包：glibc-linuxthreads。该包的版本编号与相应的 glibc 一样，因此，glibc 2.2.3 应该使用 glibc-linuxthreads 2.2.3。尽管我建议各位取回 linuxthreads 包，但是就算不使用该包，也应该能够建立 glibc。不过，请注意，例如 glibc 2.1.x，若不使用 linuxthreads 便无法顺利建立 glibc。如果使用的是 glibc 2.1.x 而且想使用 DES 加密功能，别忘了还需要下载 glibc-crypt 这个附加包。

事实上，表 4-2 一点都不完整，表中没有列出的许多其他组合也可能运行地相当好。你可以随意尝试比表中所列还新的版本并且使用前文所提到的方法：一开始使用最新的版本，需要的时候，依次使用较旧的版本。最起码，还可以回头使用表 4-2 提供的组合。

每当你发现到一个可以编译成功的新版本组合，务必测试其产生的工具链是否可以使用。有些版本的组合或许可以编译成功，但是使用的时候仍旧会失败。例如，可以在 x86 主机上使用 gcc 2.95.3 针对 PPC 目标板将 glibc 2.2.3 编译成功。然而，这样却会产生有问题的链接库，把它使用在目标板上将会造成内存转储。此时，我们只要回头使用 glibc 2.2.1，便能建立出可用的 C 链接库。

有时你也会遇到这样的情况，某种版本组合可以在某个处理器系列中特定的处理器上正常运行，却不能在相同系列的其他处理器上正常运行。例如，glibc 2.2 之前的版本，可以正常运行在大部分的 PPC 处理器上，但是 MPC8xx 系列的处理器例外。问题出在，此类的 glibc 会假定所有的 PPC 处理器都具备 32-byte 的高速缓存线，但是 MPC8xx 系列的处理器却只有 16-byte 的高速缓存线。glibc 2.2 修正了这个问体，它会假定所有的 PPC 处理器都具备 16-byte 的高速缓存线。

接下来将会针对以下组合：PPC 主机、i386 目标板、binutils 2.10.1、gcc 2.95.3 和 glibc 2.2.3，分节描述如何建立 GNU 工具链。这个组合事实上是表 4-2 的第三个条目。

建立前的准备工作

欲建立跨平台开发工具链，将会需要原生的工具链。大多数的主流发行套件都会附带此工具链。如果它并未安装在工作站上，或是如果为了节省空间而选择不要安装，此刻你必须以发行包适用的程序进行安装。例如，使用的是 Red Hat 发行包，就必须安装适当的 RPM 包。

你还需要为主机准备一组有效的内核头文件。这些头文件通常必须放在 `/usr/include/linux`、`/usr/include/asm` 和 `/usr/include/asm-generic` 等目录中，而且应该用来编译安装在你系统上的原生 glibc。在较旧的发行包以及某些安装中，这些目录实际上是 `/usr/src/linux` 目录里某些目录的符号链接。而 `/usr/src/linux` 目录本身则是发行包安装的内核源码的符号链接。如果发行套件使用的是较旧的设置，而且你有更新内核或是修改 `/usr/src/`

linux 目录的内容，请确定 */usr/src/linux* 的符号链接是否正确，这样 */usr/include* 里的符号链接才会指向你用来建立原生 *glibc* 的内核。然而，对新近的发行套件而言，*/usr/include/linux*、*/usr/include/asm* 和 */usr/include/asm-generic* 的内容已经和 */usr/src/linux* 的内容无关了，因此内核更新之后应该不会造成此类问题。

建立程序概述

拥有了需要用到的工具后，接着让我们来看看用来建立工具链的程序。此程序可分成五大步骤：

1. 内核头文件的设置
2. 二进制工具程序的设置
3. 引导编译器的设置
4. C 链接库的设置
5. 完整编译器的设置

看到这些步骤，注意到的第一件事情或许是，编译器看来好像要建立两次。这是正确且必要的，因为 *gcc* 支持的某些语言，例如 *C++*，*glibc* 也需要支持。因此，第一次建立的编译器只支持 *C*。一旦 *C* 链接库准备好之后，就可以建立支持 *C++* 的完整编译器。

尽管我将内核头文件设置列为第一个步骤，但却要等到设置 *C* 链接库的时候才会使用头文件。因此，可以改动这五个步骤的顺序，将“内核头文件设置”排在“*C* 链接库设置”之前。然而，如果将工作空间目录的安排考虑在内，将会发现这五个步骤如上所示的顺序比较恰当。

显然每个步骤都有自己的许多事要做。但是经过归纳却有许多相似之处。大部分的工具链建立步骤多半会执行以下的动作：

1. 解包
2. 为跨平台开发设定包的配置
3. 建立包
4. 安装包

有些工具链建立步骤所执行的动作跟这个顺序有些不同。例如，“内核头文件的设置”并不需要我们建立或安装内核。事实上，关于设定、建立和安装内核的许多讨论将保留到第五章再进行。此外，因为 *gcc* 包已经在“引导编译器的设置”时解开了，所以“完整编译器的设置”可以省掉解开 *gcc* 包的動作。

工作空间的设置

根据我前文建议的工作空间目录安排方式，工具链将会建立在`${PRJROOT}/build-tools`目录中，而建立的组件则会被安装到`${PRJROOT}/tools`目录中。为此，我们需要定义一些额外的环境变量。这些环境变量会用到已经定义的环境变量，可简化建立的程序。承前例，下面是用来建立新环境变量的 *develdaq* 命令脚本：

```
export PROJECT=daq-module
export PRJROOT=/home/karim/control-project/${PROJECT}
export TARGET=i386-linux
export PREFIX=${PRJROOT}/tools
export TARGET_PREFIX=${PREFIX}/${TARGET}
export PATH=${PREFIX}/bin:${PATH}
cd $PRJROOT
```

其中，TARGET 变量用来定义目标板的类型，将会根据此目标板类型建立工具链。表 4-3 提供了 TARGET 变量其他的可能值。请注意，目标板定义跟主机类型无关。目标板定义根据硬件本身以及其上所使用的操作系统（即 Linux）。同时请注意，当我们更换目标板时，只需要修改 TARGET 变量。即使你已经为之前的目标板编译好完整的 GNU 工具链，修改 TARGET 变量之后仍需要重新建立一次。TARGET 变量值的完整清单可参考 glibc 源码包附带的手册。

表 4-3: TARGET 变量值范例

实际的目标板	TARGET 变量值
PowerPC	powerpc-linux
ARM	arm-linux
MIPS (big endian)	mips-linux
MIPS (little endian)	mipsel-linux
SuperH 4	sh4-linux

PREFIX 变量为组件配置命令脚本提供了一个指针，指向目标板工具程序将被安装的目录。而 TARGET_PREFIX 变量则会指向与目标版相关的头文件和链接库将被安装的目录。欲取用刚安装好的工具程序，我们还需要修改 PATH 变量，让它指向二进制文件（可执行文件）将被安装的目录。

有些人宁可将 PREFIX 变量设成 */usr/local*，因为工具程序和链接库会被安装在 */usr/local* 目录里，所以任何用户都可以取用。然而，我发现这么做通常不太有用，因为即使不同的计划使用相同的目标架构，也可能需要不同的工具链配置。

如果需要替整个开发团队设置工具链，但不想通过 */usr/local* 目录来共享工具程序和链接

库，建议可让一位开发者在 */home* 目录底下让所有项目成员共享的目录中建立工具链，正如前文所述。如果 */home* 目录下没有可供所有项目成员共享的目录，那么这位开发者可以在自己的工作站上 */opt* 目录下某个目录中建立工具链，然后将成果共享给其他成员。他可以使用传统的共享机制，如 NFS，或是将工具链以 tar-gzip 的方式打包放在 FTP 服务器上。每个开发者使用此包时，应该根据当初建立工具链存放的文件系统层次放在相同的位置上，这样方能正常运行。也就是说，如果工具链是在 */opt* 目录中建立的，每个开发者都应该将工具链放在 */opt* 目录中。

如果选择将 `PREFIX` 设成 */usr/local*，还必须以 root 的身份执行后面提到的命令，不过这么做要冒一定的风险。你可能还需要设定 */usr/local* 目录的使用权限，好让你自己或用户组能够不用 root 特权就能执行命令。

请注意，`TARGET_PREFIX` 变量会被设成 `${PREFIX}/${TARGET}`，这是个因目标板而异的目录。如果将 `PREFIX` 变量设成 */usr/local*，之后若相继针对不同的目标板安装开发工具链，结果最后安装的链接库和头文件将会放在跟前一次安装不同的地方。

不管 `PREFIX` 变量被设成什么值，将 `TARGET_PREFIX` 变量设成 `${PREFIX}/${TARGET}` 的好处是，它会跟着变。因此，建议将 `TARGET_PREFIX` 变量设成设成此值。如果变更了 `TARGET_PREFIX` 变量的值，可能需要修改接下来的说明。

再一次提醒，如果不想直接移往项目目录，可以删除命令脚本中的 `cd` 命令。

准备 build-tools 目录

此刻，*build-tools* 目录中应该包含用来建立工具链的各种包。每个包都一样，当你从包中取出文件时，就应该为它建立一个新的目录。这个新目录应该包含用来建立包的完整源码以及各种 Makefile。尽管可以在存放源码的目录里建立包，不过我强烈建议应该在源码目录以外的目录建立每个包，正如 FSF 安装手册中的建议那样。

如果只会 *configure; make; make install* 这一套的话，要在源码目录以外的目录建立包，似乎让你难以招架，没关系，稍后我会告诉你方法。不过，首先必须建立目录，以便存放欲建立的包。我们应该为每个工具链组件建立一个目录。因此需要建立四个目录，分别用来存放：二进制工具程序、引导编译器、C 链接库、完整编译器。我们可以使用如下的命令来建立必要的目录：

```
$ cd ${PRJROOT}/build-tools
$ mkdir build-binutils build-bootstrap gcc build-glibc build-gcc
```

现在我们可以检查 *build-tools* 目录中包含的包以及建立的目录（因为版面的关系，此范例的最后一行被截掉了）：

```
$ ls -l
total 35151
-rw-r--r-- 1 karim karim 7284401 Apr 4 17:33 binutils-2.10.1.tar.gz
drwxrwxr-x 2 karim karim 1024 Apr 4 17:33 build-binutils
drwxrwxr-x 2 karim karim 1024 Apr 4 17:33 build-boot-gcc
drwxrwxr-x 2 karim karim 1024 Apr 4 17:33 build-gcc
drwxrwxr-x 2 karim karim 1024 Apr 4 17:33 build-glibc
-rw-r--r-- 1 karim karim 12911721 Apr 4 17:33 gcc-2.95.3.tar.gz
-rw-r--r-- 1 karim karim 15431745 Apr 4 17:33 glibc-2.2.3.tar.gz
-rw-r--r-- 1 karim karim 215313 Apr 4 17:33 glibc-linuxthreads-2.2.3.t
```

似乎一切都准备好了，现在可以开始建立工具链。

资源

实际建立工具链之前，先让我们来看一下，当你在建立的过程中遇到问题时，可以找到哪些有用的资源。

首先，每个包本身都会附带自己的文件。尽管 binutils 附带的只是一些与安装有关的文件，不过至少可以试试看是否可以找到引发问题的原因。然而，gcc 和 glibc 就附带了很详细的文档。在 gcc 包中，可以找到一个 FAQ 文件以及一个 *install* 目录，内含如何设定与安装 gcc 的各种指示。这包括对建立时的配置选项的大量说明。同样地，glibc 包中也可以找到 FAQ 和 INSTALL 文档。INSTALL 文档中包含建立时的配置选项、安装程序以及编译工具的版本建议。

此外，或许你想要使用一般的搜寻引擎（如 Google）来找寻是否已经有其他开发者遇到并且解决了跟你类似的问题。通常，使用一般的搜寻引擎将会是解决 GNU 工具链建立问题最有效的方法。

就交叉编译而言，可以找到两份 CrossGCC FAQ，分别为 Scott Howard 版和 Bill Gatliff 版。Scott Howard 版的 CrossGCC FAQ 可以从 <http://www.sthoward.com/CrossGCC/> 获得。不过这份 FAQ（常见问题解答）已经有点旧了。Bill Gatliff 版的 CrossGCC FAQ 可以从 <http://crossgcc.billgatliff.com/> 获得。

尽管 Scott Howard 版 CrossGCC FAQ 已经旧了，不过它的内容并不局限在 Linux，而且还试图为 GNU 工具链所能建立的任何平台提供一般性的说明，其所提供的信息事实上很难在别处找到。例如，它的内容涵盖了所谓的 Canadian Crosses 技术（注 3），也就是在另一个平台建立跨平台开发工具的技术。比如说，在 PPC 工作站上为 ARM 目标板和 i386 主机建立跨平台开发工具。

注 3：事实上 Canada（加拿大）有三大族群（英语系人口、法语系人口、讲母语的原住民），每当提到一个名称的时候，的确需要这样的程序。

如同 Scott Howard 版的 FAQ、Bill Gatliff 版的 FAQ 也没有局限在 Linux 上。除了 FAQ，Bill Gatliff 还很积极地维护了一个 CrossGCC Wiki 网站，该网站提供了与各种跨平台开发问题有关的信息，包括教学文件、相关文章的链接，以及跟 GNU 工具链内部组件有关的说明。因为这是个 Wiki 网站，如果想要修改或为该网站做贡献的话，可以在该网站注册。你可以通过前面提到的网址找到这个 Wiki 网站。

这两份 FAQ 都有提供自动建立工具链的命令脚本。类似的命令脚本也可以在许多其他网站找到。或许你有兴趣想要看一下这些命令脚本，不过在你对整个步骤有全盘的了解之前，我是不会对任何命令脚本提出说明的。

最后要说的是，Red Hat 在 <http://sources.redhat.com/ml/crossgcc/> 维护了一个 crossgcc 邮件论坛。如果有什么疑难杂症，会发现这个邮件论坛相当有用，因为此论坛上对跨平台开发工具链建立程序很有经验的人还不少。通常只要搜寻或浏览已归档的讨论内容，就可以立即找到问题的答案。

关于预先建好的跨平台工具链

网络上或商业上有许多预先建好的跨平台工具链。因为我并不知道它们的实际建立过程，所以无法对这些包提供任何建议。如果为了方便，可能会选用这类包，而舍弃本书所提到的步骤。如果真是这样的话，务必获得能够说明这些包是如何设定与建立的文件。最重要的是，确定你知道每个包的版本、要做哪些修补（如果需要的话），以及需要进行修补的时候到哪里获得补丁。

内核头文件的设置

正如前文所述，内核头文件设置是建立工具链的第一步。在这个范例中，我们使用的是 2.4.18 版的内核，不过你可以使用适合目标板使用的任何其他版本。我们将会在第五章深入探讨如何选用内核。

选好内核版本后，首先要将该内核的副本下载至用来存放内核的目录。就前文建议的工作空间结构而言，此副本应该存入 `$(PRJROOT)/kernel`。所有的 Linux 内核都可以从位于 <http://www.kernel.org/> 的主要内核库或任何其他镜像网站（如各国的镜像网站，注 4）获得。不过，有的网站提供的内核比较适合用在特定目标板上，第五章将会有所说明。

注 4：有些国家在当地有镜像网站，位于这些国家的用户或许可在当地下载内核。这些镜像网站的网址通常具有 <http://www.COUNTRY.kernel.org/> 的形式。例如，<http://www.it.kernel.org/> 和 <http://www.cz.kernel.org/>。

现在，内核的每个版本都会包含两种档案：经tar-gzip包装的档案（扩展名为*.tar.gz*）和经tar-bzip2包装的档案（扩展名为*.tar.bz2*）。这两种档案包含的都是相同的内核，不过与经tar-gzip包装的档案相比，经tar-bzip2包装的档案比较小，所以下载时间也比较短。

内核下载至你选定的目录后，可以使用适当的命令将它解开。就这个例子来说，我们会使用如下其中一道命令（这取决于我们所下载的档案）：

```
$ tar xvfz linux-2.4.18.tar.gz
```

或：

```
$ tar xvjf linux-2.4.18.tar.bz2
```

一些版本较旧的tar程序并不支持j选项，或许你在使用tar命令之前，需要先使用bzip2 -d 或 bunzip2 将包解压缩。

对版本编号在2.4.18（含）以前的内核而言，tar命令会建立一个名为linux的目录，此目录内含从包装文件取出的档案。然而，自2.4.19开始，该目录的名称会附加上版本的编号。因此，Linux 2.4.19的源码会直接存入linux-2.4.19目录中。这样可以避免旧版内核被新版内核覆盖掉。如果要使用的内核版本其编号低于2.4.19，建议将现有的linux目录更名，以免原有的内容遭到覆盖：

```
$ mv linux linux-2.4.18
```

一个内核版本会因为存放之前版本的目录未被更名而覆盖另一个内核版本，这是一般人常犯的错误，因此我再三强调，如果需要从包中取出内核源码，就必须将现有的linux目录更名。

内核源码取出后，接下来进行配置设定：

```
$ cd linux-2.4.18
$ make ARCH=i386 CROSS_COMPILE=i386-linux- menuconfig
```

这将会在控制台上显示一份菜单，可以藉此选择内核的配置。不想使用menuconfig的话，还可以使用config或xconfig。前者会要求你在命令行上依次为每个可能的配置选项提供一个答案。而后者则会提供一个X Window对话框，这是公认最直观的内核配置设定方式。然而，使用xconfig的时候必须小心，因为它可能无法设定有些配置选项，而且会忘记产生我所描述的程序所需要的头文件。使用config也可能会造成有些头文件没有被建立的结果。要检查某个内核配置是否成功建立适当的头文件，在你完成配置设定程序后，可以检查内核源码树中是否存在include/linux/version.h文件。如果找不到，下面提到的指令会在首次需要用到内核头文件的时候失败；通常在glibc编译期间。

你可能注意到了，变量 ARCH 和 CROSS_COMPILE 的值与目标板的架构类型有关。倘若我们使用的是 PPC 目标板和 i386 主机，则结果会变成 ARCH=ppc 以及 CROSS_COMPILE=powerpc-linux-。（在 CROSS_COMPILE=powerpc-linux- 中，结尾的连字符并非偶然。）严格来说，为目标板建立内核时，并不需要设定 CROSS_COMPILE 变量。例如，我之前提到的各种目标板配置，通常都不需要做这个设定。事实上，只有在由于内核 Makefile 的规则，导致必须实际进行程序代码交叉编译时，才需要这么做。然而，在本书中，只要一提到为目标板建立内核的命令，不管是否必要，我都会做此设定，以便强调其重要性。实际应用时，可依需要决定是否做此设定。

我将会在第五章深入探讨内核的配置。如果对内核的配置并不熟悉，可以先大略看一下第五章。此刻我们需要设定的配置选项中，最重要的是处理器和系统的类型。尽管在继续下去之前最好能完成内核配置的全部设定，不过只要设好处理器和系统的类型，通常就能够产生工具链建立时需要的头文件。

设好内核配置之后，可用右键选择 Exit 条目，离开该菜单。然后配置工具程序会问你是否要保存配置，确认之后，接着会写入内核配置，并且建立适当的文件和链接。

现在，可以建立工具链所需要的 *include* 目录，并将内核头文件复制过去：

```
$ mkdir -p ${TARGET_PREFIX}/include
$ cp -r include/linux/ ${TARGET_PREFIX}/include
$ cp -r include/asm-i386/ ${TARGET_PREFIX}/include/asm
$ cp -r include/asm-generic/ ${TARGET_PREFIX}/include
```

不要忘了，我们现在使用的是 PPC 主机和 i386 目标板的组合。因此，*asm-i386* 目录将会包含该目标板专属的头文件，而不是主机专属的头文件。倘若你使用的是 PPC 目标板，则必须将 *asm-i386* 替换成 *asm-ppc*。

请注意，不必在每次重新设定内核配置之后重建工具链。工具链只需要一组可供目标板使用的有效头文件即可，这些头文件在早先的程序中就已经提供了。今后你可能会选择重新设定内核的配置，或是使用另一个内核，这都不会对工具链造成任何的影响，除非你变更了处理器或系统的类型。

binutils 的设置

binutils 包中的工具常用来操作二进制目标文件。该包中最重要的两个工具就是 GNU 汇编器 *as* 和链接器 *ld*。表 4-4 完整地列出了 binutils 包中可以找到的工具。

表 4-4: binutils 包中可以找到的工具

工具	用途
<i>as</i>	GNU 汇编器
<i>ld</i>	GNU 链接器
<i>gasp</i>	GNU 汇编器预处理器
<i>ar</i>	用来创建与操作档案文件的内容
<i>nm</i>	列出目标文件中的符号列表
<i>objcopy</i>	复制并转换目标文件
<i>objdump</i>	显示目标文件内容有关的信息
<i>ranlib</i>	为档案文件的内容产生索引
<i>readelf</i>	显示 ELF 格式的目标文件内容有关的信息
<i>size</i>	列出目标文件中各区段的大小
<i>strings</i>	列出目标目标文件中的可打印字符
<i>strip</i>	除去目标文件中的符号
<i>c++filt</i>	将因为 c++ 函数的重载而损坏的低级汇编语言标号转换成用户层的名称
<i>addr2line</i>	将所指定的地址转换成源文件中的行号

请注意，尽管 *as* 支持许多种处理器架构，但是它并不需要让不同的架构使用不同的语法。*as* 支持的语法与机器无关，此灵感来自 BSD 4.2 的汇编语言。

设置 binutils 包的第一步，就是从我们前面下载的档案文件中取出源码：

```
$ cd $(PRJROOT)/build-tools
$ tar xvzf binutils-2.10.1.tar.gz
```

这将会产生一个包含该包内容的 *binutils-2.10.1* 目录。现在我们可以移往 *build-binutils* 目录，以便进行建立程序的第二个部分，为跨平台开发设定包的配置：

```
$ cd build-binutils
$ ../binutils-2.10.1/configure --target=$TARGET --prefix=${PREFIX}
Configuring for a powerpc-unknown-linux-gnu host.
Created "Makefile" in /home/karim/control-project/dag-module/build-...
Configuring intl...
creating cache ../config.cache
checking for a BSD compatible install... /usr/bin/install -c
checking how to run the C preprocessor... gcc -E
checking whether make sets ${MAKE}... yes
checking for gcc... gcc
checking whether the C compiler (gcc -g -O2 -W -Wall ) works... yes
checking whether the C compiler (gcc -g -O2 -W -Wall ) is a cross-c...
```

```
checking whether we are using GNU C... yes
checking whether gcc accepts -g... yes
checking for ranlib... ranlib
checking for POSIXized ISC... no
checking for ANSI C header files... yes
...
```

以上只列出了 *configure* 命令脚本的部分输出。事实上，此命令脚本会继续在控制台上列出类似的信息，直到它为了包中每个工具的编译设定好配置为止。这可能要等上一两分钟的时间，不过相对来说算是短的了。

configure 执行的时候会检查主机上是否存在某些资源，并且会为包中每个工具程序产生适当的 Makefile。如果不是在包含 *binutils* 源码的目录中执行 *configure* 命令，那么 *configure* 命令的执行结果将会放在你执行命令当时所处的目录之中，也就是 *build-binutils* 目录。

我们可以通过将适当的选项传递给 *configure* 来控制 Makefile 的输出。其中，*--target* 选项让我们能够指定，这是为哪个目标板建立 *binutils*。因为我们已经在 *TARGET* 环境变量中指定好了目标板的名称，所以我们将以这个变量进行指定。而 *--prefix* 选项让我们能够指定安装目录。同样地，因为我们已经在 *PREFIX* 环境变量中指定好了安装目录，所以我们将以这个变量进行指定。准备好 Makefile 后，接下来就是建立实际的工具：

```
$ make
```

实际建立 *binutils* 的过程，可能要耗时 10 到 30 分钟，这取决于你使用的硬件。以 400 MHz 的 PowerBook 为例，要为这里使用的 i386 目标板建立 *binutils*，至多 15 分钟。建立期间你可能会看到一些警告信息，不过你可以忽略这些警告信息，除非你是个 *binutils* 开发者。

包建立好之后，接着安装 *binutils*：

```
$ make install
```

binutils 会被安装到 *PREFIX* 变量指定的目录。你可以借着列出适当的目录来检查安装结果是否正确：

```
$ ls ${PREFIX}/bin
i386-linux-addr2line  i386-linux-ld          i386-linux-readelf
i386-linux-ar         i386-linux-rm          i386-linux-size
i386-linux-as         i386-linux-objcopy     i386-linux-strings
i386-linux-c++filt    i386-linux-objdump     i386-linux-strip
i386-linux-gas        i386-linux-ranlib
```

请注意，每个工具的文件名的前缀都是我们前面为 *TARGET* 变量设定的值。例如，倘若

目标板是 `powerpc-linux`，那么这些工具的文件名前缀就会是 `powerpc-linux-`。这样，为目标板建立应用程序的时候，我们就可以根据目标板类型找到正确的工具程序。

有些工具程序的副本不会前置目标板类型的名称，这类程序将会被安装到 `${PREFIX}/${TARGET}/bin` 目录。因为稍后在 C 链接库的建立过程中，这个目录将会用来安装目标板二进制文件，我们将需要把主机二进制文件移往更合适的目录。我们稍后再讨论这个问题。

引导编译器的设置

与 `binutils` 套件相比，`gcc` 套件只包含了一个工具程序（即 GNU 编译器），不过它还支持一些组件，如运行时库。在目前这个阶段，我们会建立引导编译器，该编译器只支持 C 语言。等到 C 链接库编译好之后，我们会重新编译 `gcc` 并提供完整的 C++ 支持。

同样地，我们一开始会从前面下载的包中取出 `gcc` 包的内容：

```
$ cd ${PRJROOT}/build-tools
$ tar xvzf gcc-2.95.3.tar.gz
```

这会产生一个包含包内容的 `gcc-2.95.3` 目录，接下来在我们为引导编译器准备的目录里设定建立引导编译器的配置：

```
$ cd build-boot-gcc
$ ../gcc-2.95.3/configure --target=${TARGET} --prefix=${PREFIX} \
> --without-headers --with-newlib --enable-languages=c
```

这会列出与前面我们讨论 `binutils` 配置工具（命令脚本）时类似的输出。这里也一样，`configure` 会检查有哪些可用的资源，并且建立 `Makefile`。

这里为 `configure` 指定的 `--target` 和 `--prefix` 选项，其目的如同前面探讨 `binutils` 时所做的说明，就是分别用来指定目标板类型及安装目录。此外，我们还使用了建立引导编译器时需要的选项。

因为这是个交叉编译器，还不需要目标板的系统头文件，所以我们需要使用 `--without-headers` 选项。我们还要以 `--with-newlib` 选项来告诉配置工具不要使用 `glibc`，因为 `glibc` 尚未针对目标板完成编译的动作。然而，这个选项并无法强迫我们使用 `newlib` 作为目标板的 C 链接库。这只是让 `gcc` 能够正确编译而已，稍后我们还可以随意选择任何其他 C 链接库。

`--enable-languages` 选项用来告诉配置命令脚本，我们想让产生的编译器支持哪些程序语言。因为这是个引导编译器，所以我们只需支持 C 语言。

因设置而定，或许你想要为目标板使用额外的选项。关于 *configure* 可用选项的完整清单，请参考 gcc 包所提供的安装说明。

准备好 Makefile 之后，接着建立编译器：

```
$ make all-gcc
```

引导编译器的编译时间与 binutils 的差不多。同样，也会在编译期间看到警告信息，也可以忽略不管。

编译完成之后，接着安装 gcc：

```
$ make install-gcc
```

引导编译器的安装目录跟 binutils 一样，可以重新列出 *\${PREFIX}/bin* 的内容来检查一番。如同工具程序，编译器的文件名的前缀也是目标板的名称，就此例来说，编译器的文件名就是 *i386-linux-gcc*。

C 链接库的设置

glibc 包由许多链接库组成，它是我们跨平台开发工具链中，建立过程最麻烦、建立程序最冗长的包。glibc 是极其重要的一个软件组件，目标板必须靠它来执行或开发大部分（如果不是所有）的应用程序。请注意，尽管 glibc 套件常被称为 C 链接库（GNU 自己的文件里仍存在此情况），不过 glibc 实际上会产生许多链接库，其中之一便是 C 链接库 libc。我们将在第六章说明如何让 glibc 产生所有的链接库。在此之前，我将会继续混用 C 链接库与 glibc 这两个名称。

正如之前的包，我们一开始会从前面所下载的包中取出 C 链接库的源码：

```
$ cd ${PRJROOT}/build-tools  
$ tar xvzf glibc-2.2.3.tar.gz
```

这会产生包含包内容的 *glibc-2.2.3* 目录。除了取出 C 链接库的源码，我们还会基于本章前面提到的理由，取出 linuxthreads 包的源码：

```
$ tar -xvzf glibc-linuxthreads-2.2.3.tar.gz --directory=glibc-2.2.3
```

接着，我们可以移往 *build-glibc* 目录，开始建立 C 链接库：

```
$ cd build-glibc  
$ CC=i386-linux-gcc ../glibc-2.2.3/configure --host=$TARGET \  
> --prefix="/usr" --enable-add-ons \  
> --with-headers=${TARGET_PREFIX}/include
```

使用 gcc 3.2 及以上的版本

在写作本书时，前一节的说明对 gcc 3.2 和 3.2.1 并不适用，因为 `--without-headers` 配置选项有问题而且尚未修正。要解决此问题，在编译引导编译器之前必须先安装适当的 glibc 头文件。此处将会告诉你如何安装这些头文件，但不会详细描述每个命令选项，因为这些命令选项的说明已经涵盖在前一节及下一节中了。此处，我们将会使用 binutils 2.13.2.1、gcc 3.2.1 和 glibc 2.3.1。请注意，原生的 gcc 3.2 必须已经安装在主机上，才能进行以下步骤。

首先，如同设置 C 链接库一样，我们必须从 glibc 包及其附加包取出源码：

```
$ cd ${PRJROOT}/build-tools
$ tar xvzf glibc-2.3.1.tar.gz
$ tar -xvzf glibc-linuxthreads-2.3.1.tar.gz --directory=glibc-2.3.1
```

接着，我们必须设定 glibc，并且安装它的头文件：

```
$ mkdir build-glibc-headers
$ cd build-glibc-headers
$ ../glibc-2.3.1/configure --host=$TARGET --prefix="/usr" \
> --enable-add-ons --with-headers=${TARGET_PREFIX}/include
$ make cross-compiling=yes install_root=${TARGET_PREFIX} \
> prefix="" install-headers
```

因为我们并未将 CC 变量设成指向既有的交叉编译器，所以我们必须将 `cross-compiling` 变量设成 `yes`，这样建立 glibc 的命令脚本才不会建立原生的链接库。最后，使用 `install-headers` 这个 Makefile target 要求 `make` 安装头文件。

接下来，我们会产生一个假的 `stubs.h` 文件，因为 gcc 在建立 glibc 时需要这个文件（经交叉编译的 glibc 在安装期间会产生该文件）：

```
$ mkdir -p ${TARGET_PREFIX}/include/gnu
$ touch ${TARGET_PREFIX}/include/gnu/stubs.h
```

最后，我们可以建立引导程序 gcc 编译器：

```
$ cd ${PRJROOT}/build-tools/build-boot-gcc
$ ../gcc-3.2.1/configure --target=$TARGET --prefix=${PREFIX} \
> --disable-shared --with-headers=${TARGET_PREFIX}/include \
> --with-newlib --enable-languages=c
$ make all-gcc
```

除了前一节用到的选项，我们还以 `--disable-shared` 配置选项来告诉建立命令脚本，不要产生共享的 gcc 链接库。如果不使用这个选项，gcc 3.2 将无法建立成功。

一旦引导编译器安装好之后，接下来的步骤就是建立与安装 GNU 工具链的其他组件，方法如同本章之前所做的说明。

请注意，这个配置命令跟前面的有些不同。首先，我们会在调用 *configure* 之前，事先执行 *CC=i386-linux-gcc*。目的就是环境变量 *CC* 设成 *i386-linux-gcc*。因此，这里用来建立 C 链接库的编译器，将会是我们刚才建立的引导编译器。同时，我们还会以 *--host* 选项来取代 *--target* 选项，因为此链接库将会在目标板上执行，而不是在“建立系统”（注 5）上。换言之，从链接库的观点来看，主机就是我们的目标板，相对而言，我们前面所建立的工具程序，将会在“建立系统”上执行。

尽管此处仍旧使用了 *--prefix* 选项，不过其目的在于告诉配置命令脚本，一旦在目标板的根文件系统上时，链接库组件的位置。glibc 编译期间会将此位置在 glibc 组件中硬编码，并且在运行时使用。如同 glibc 源码目录中 *INSTALL* 文件所做的说明，Linux 系统将会到 */lib* 和 */usr/lib* 目录寻找 glibc 组件。将 *--prefix* 设成 */usr*，配置命令脚本会认可这种设置，并将相关的目录路径在 glibc 组件中硬编码。结果，举例来说，动态链接器会到 */lib* 寻找共享链接库，在任何 Linux 系统中这都是存放此类链接库的适当位置，正如我们在第六章所见。然而，我们将不会让“建立命令脚本”把链接库安装到“建立系统”的 */usr* 目录。本节稍后我们在执行 *make install* 的时候，将会让这个安装目录的设定失效。

我们还会以 *--enable-add-ons* 选项来要求配置命令脚本使用我们下载的附加包。我们可藉此指定实际会用到的附加包清单，因为只会用到 *linuxthreads* 附加包，所以可用 *--enable-add-ons=linuxthreads* 选项来设定它。如果使用的是 glibc 2.1.x，则还必须用到 *glibc-crypt* 附加包，此时可用 *--enable-add-ons=linuxthreads,crypt* 选项设定它。尽管上面这个命令并未提供附加包的完整清单，不过对大多数的 glibc 版本来说，通常不会有什么问题。

最后，我会以 *--with-headers* 选项告诉配置命令脚本，何处可以找到我们前面设置的内核头文件。如果省略该选项，它将使用由 */usr/include* 找到的头文件来建立 glibc，这是有问题的，因为它们是“建立系统”本身的头文件，而非目标板的头文件。

在链接库实际建立期间，会建立三组链接库：共享链接库、静态链接库、具有统计信息（*profiling information*）的静态链接库。如果不想使用具有统计信息的链接库，可以在建立的过程中使用 *--disable-profile* 选项告诉配置命令脚本。如果不想使用共享链接库，则可以使用 *--disable-shared* 选项。倘若目标板需要用到的应用程序不多，而且你准备静态链接所有的应用程序，或许你会想使用这个选项。然而，请三思而行，因为目标板到最后可能需要共享链接库。事实上，即使留下共享链接库也没关系，因为你仍旧可以使用静态链接的方式建立应用程序。这么做，至少在你想要改变应用程序链接方式的时候，不必重建 C 链接库。

注 5：实际上，“建立系统”就是我们的开发主机。

`--enable-static-nss`是另一个与静态和动态链接有关的选项。使用此选项会产生静态链接的 Name Service Switch (NSS) 组件。简单地说, NSS 让某些链接库组件能够在机器本地的配置中进行定制。这包括用 `/etc/nsswitch.conf` 文件来指定于运行时加载 `/lib/libnss_NSS_SERVICE` 链接库。因为这项服务被刻意设计成动态加载链接库, 它将不会允许采用静态链接的方式, 除非你强迫它这么做。因此, 如果准备以 NSS 静态链接应用程序, 则必须在执行配置命令脚本的命令行上加入 `--enable-static-nss` 选项。例如, 第十章提到的 web 服务器会用到 NSS, 如果要求链接器将 web 服务器静态链接至不允许静态链接 NSS 的 glibc, 那么 web 服务器在目标板上的功能不是不正常就是会建立失败。关于 NSS 的完整讨论请参考 glibc 使用手册。

如果正在为不支持 FPU 的目标板编译 glibc, 还可能会使用 `--without-fp` 选项, 以便在 C 链接库中建立 FPU 的仿真功能。有时, 也可能需要在用来建立链接库的 C 标志中加入 `-msoft-float` 选项。(自从 glibc 2.3) 每当你以 `--without-fp` 来设定 glibc 的时候, 至少对 PPC 来说, C 标志都会有适当的设定。

如果使用的是 glibc 2.1.x, 不过你不想下载 linuxthreads 包或 crypt 包, 请在编译 C 链接库的时候, 除掉 `--enable-add-ons` 选项并加上 `--disable-sanity-checks` 选项。否则, 配置命令脚本将会抱怨找不到 linuxthreads。然而, 请注意, 尽管 glibc 在缺乏 linuxthreads 的情况下可能会建立成功, 但是稍后为了支持 C++ 而建立完整编译器的时候, 可能会失败。

以配置命令脚本完成设定后, 接着编译 glibc:

```
$ make
```

C 链接库是个非常大的包, 它的编译过程可能要耗费几个小时, 这取决于你使用的硬件。在稍早提到 PowerBook 系统上, 大约要耗费 1 个小时。不管你使用哪种平台, 这个时候你正好可以休息一下, 吃些点心或是到屋外呼吸新鲜空气。不过, 此刻最好避免为了让计算机在前台执行其他工作, 而让编译 C 链接库的工作在后台执行。正如前面所说, C 链接库中某些组件的编译工作需要用到大量的内存空间, 如果因为缺乏可用的内存空间而导致编译失败, 可能必须从头开始重新建立链接库, 也就是先执行 `make clean` 再执行 `make`。

一旦 C 链接库建立好之后, 接着进行安装:

```
$ make install_root=${TARGET_PREFIX} prefix="" install
```

与其他套件的安装相比, C 链接库的安装将会耗费不少时间。尽管安装时间不如编译时间长, 但也要耗费 5 到 10 分钟的时间, 同样地, 这取决于你使用的硬件。

请注意，此处的安装命令与传统的 *make install* 命令不同。我们设定 *install_root* 变量，让它指向链接库组件目前所要安装的目录。这么做可以让链接库及其头文件安装到我们前面通过 *TARGET_PREFIX* 指定的与目标板有关的目录，而不是“建立系统”本身的 */usr* 目录。同时，因为之前使用 *--prefix* 选项来设定 *prefix* 变量的值，而且 *prefix* 的值会被附加到 *install_root* 的值之后，成为链接库组件的安装目录，所以我们重新设定 *prefix* 的值，让所有 *glibc* 组件直接安装在 *\$(TARGET_PREFIX)* 所指的目录下。因此，原本会被安装到 *\$(TARGET_PREFIX)/usr/lib* 目录下的 *glibc* 组件，现在会被安装到 *\$(TARGET_PREFIX)/lib*。

为目标板建立工具程序时，如果目标板的架构跟主机一样（例如，在 PPC 主机上为 PPC 目标板编译工具程序），可能会想要在所执行的 *make install* 命令行上将 *cross-compiling* 的值设成 *yes*。因为链接库的 *configure* 命令脚本在建立配置期间会发现架构一致的情况，因此 *Makefile* 并不认为你要执行交叉编译，所以 *Makefile* 会使用不同的规则，进而导致安装 C 链接库失败。

接下来是完成 *glibc* 安装的最后一个步骤：*libc.so* 文件的配置。将应用程序链接到 C 链接库期间便会用到这个文件，它实际上是一个链接命令脚本。*libc.so* 文件中指出了实际链接时需要用到的各种链接库。用上面的 *make install* 命令进行安装时，会假定链接库安装到根文件系统上，因此会在 *libc.so* 链接命令脚本中以绝对路径名称指出需要的链接库。因为我们目前将 C 链接库安装到非标准的目录中，所以必须修改链接命令脚本，让链接器能够用到的正确的链接库。这个链接命令脚本会随着 C 链接库的其他组件一起安装到 *\$(TARGET_PREFIX)/lib* 目录。

libc.so 档案最初的内容如下所示：

```
/* GNU ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily.  */
GROUP ( /lib/libc.so.6 /lib/libc_nonshared.a )
```

即使这个文件的内容跟发行包为 */usr/lib/* 目录的原生 C 链接库安装的 *libc.so* 文件不同，也会相当类似。因为目标板有时可能会需要原来的命令脚本，所以建议在修改前留下副本：

```
$ cd $(TARGET_PREFIX)/lib
$ cp ./libc.so ./libc.so.orig
```

现在你可以编辑这个文件，除掉它列举的链接库的绝对路径。基本上，将需要除掉所有链接库文件名中的 */lib/* 字样。新产生的 *libc.so* 文件，看起来应该像这样：

```
/* GNU ld script
   Use the shared library, but some functions are only in
```



```
the static library, so try that secondarily.  */
GROUP ( libc.so.6 libc_nonshared.a )
```

将该文件列举的链接库的绝对路径除掉，可以迫使链接器使用与*libc.so*命令脚本放在同一个目录的链接库，这正是目标板要使用的链接库，而不是去寻找供主机使用的原生链接库。

完整编译器的设置

我们现在可以为目标板安装支持C和C++的完整编译器。因为我们已经在“引导编译器的设置”中从包中取出了编译器的源码，所以不必重复这个步骤。大体而言，完整编译器的建立程序比引导编译器简单多了。

首先移到 *build-tools/build-gcc* 目录，然后进行配置设定的工作：

```
$ cd ${PRJROOT}/build-tools/build-gcc
$ ../gcc-2.95.3/configure --target=$TARGET --prefix=${PREFIX} \
> --enable-languages=c,c++
```

我们在此处用到的选项的意义如同建立引导编译器用到的选项。然而，请注意，此处的选项比较少，而且除了C之外，我们现在还增加了对C++的支持。倘若你正如前面所说的那样把 *TARGET_PREFIX* 设成 *\${PREFIX}/\${TARGET}* 以外的值，你将需要使用 *--with-headers* 和 *--with-libs* 选项来通知配置命令脚本到哪里寻找glibc安装的头文件和链接库。

完整编译器的配置设定好之后，就可以进行建立的步骤了：

```
$ make all
```

建立过程耗费的时间会比引导编译器还长一点。同样地，可能也会看到可以忽略的警告信息。请注意，此处的建立命令使用的是 *all*，不像引导编译器使用的是 *all-gcc*。这么做，将会建立gcc包的所有组件，包括C++运行时链接库。

如果并未将之前所说的*libc.so*链接命令脚本设定好，则建立工作在“运行时链接库”编译期间将会失败。同样地，如果在C链接库设置期间并未安装linuxthreads包，则某些版本的gcc可能会编译失败。例如，gcc 2.95.3将会因为没有linuxthreads而导致建立失败。

完整编译器建立成功后，便可进行安装的步骤：

```
$ make install
```

这将会安装完整编译器，并将之前安装的引导编译器覆盖掉。请注意，此处的建立命令使用的是 *install*，不像引导编译器使用的是 *install-gcc*。同样地，这是因为我们现在不仅要安装gcc还包括它支持的组件。

完成工具链的设置

整个跨平台开发工具链现在设置好了，几乎到了已经可以使用的地步。只差最后几个检查的工作要做。

首先，让我们检查 *tools* 目录中安装了哪些文件，以及往后如何使用这些文件。表 4-5 列出了 *tools* 目录中第一层的子目录：

表 4-5: `${PRJROOT}/tools` 目录的内容

目录	内容
<i>bin</i>	交叉开发工具
<i>i386-linux</i>	目标板专用文件
<i>include</i>	供交叉开发工具使用的头文件
<i>info</i>	gcc 的 info 文件
<i>lib</i>	供交叉开发工具使用的链接库
<i>man</i>	交叉开发工具的在线说明文件
<i>share</i>	交叉开发工具与链接库共享的文件。这目录目前是空的

其中有两个目录最重要：*bin* 和 *i386-linux*。*bin* 目录包含了交叉开发工具链所有的工具程序，我们将会在主机上使用这些工具来为目标板开发应用程序。*i386-linux* 目录中的软件组件都会应用在目标板上。它的主要内容是目标板的头文件和运行时链接库。表 4-6 列出了 *i386-linux* 目录中的第一层子目录：

表 4-6: `${PRJROOT}/tools/i386-linux` 目录的内容

目录	内容
<i>bin</i>	与 glibc 有关的目标板二进制文件和命令脚本
<i>etc</i>	目标板的 <i>/etc</i> 目录中应该存放的文件。只包含 <i>rpc</i> 文件
<i>include</i>	供目标板用来建立应用程序的头文件
<i>info</i>	glibc 的 info 文件
<i>lib</i>	目标板的 <i>/lib</i> 目录
<i>libexec</i>	二进制辅助程序。此目录只包含 <i>pt_chown</i> 文件，大多数的目标板都不会用到此文件
<i>sbin</i>	目标板的 <i>/sbin</i> 目录
<i>share</i>	与国际化有关的子目录和文件
<i>sys-include</i>	供 gcc 配置命令脚本复制的目标板头文件，glibc 并未将这些目标板头文件安装到 <i>include</i> 目录

在 *i386-linux* 目录中有两个子目录最重要：*include* 和 *lib*。*include* 目录包含的头文件会被用来建立目标板的应用程序。而 *lib* 目录则包含了供目标板使用的运行时链接库。

请注意，*lib* 目录包含了许多大型的链接库。要加入此目录需要大约 80M 的存储空间。大多数的嵌入式系统通常无法提供这样大的存储空间。如同我们将在“C 链接库的替代品”一节所见，事实上存在可以取代 *glibc* 的其他链接库。此外，我们还会在第六章看到，若想将链接库的数量和大小降到最低有哪些方案可供选择。

如同我前面所说，一些文件名前缀中没有目标板名称的主机工具程序会被安装到 *\$(PREFIX)/\$(TARGET)/bin* 目录。因为此目录现在包含了 C 链接库建立过程中安装的目标板二进制文件，强烈建议将主机二进制文件移往较适合的目录存放。需要移出的工具程序包括 *as*、*ar*、*gcc*、*ld*、*nm*、*ranlib* 和 *strip*。你可以用 *file* 命令来验证这些文件的确是主机二进制文件：

```
$ cd $(PREFIX)/$(TARGET)/bin
$ file as ar gcc ld nm ranlib strip
as:      ELF 32-bit MSB executable, PowerPC or cisco 4500, version 1...
ar:      ELF 32-bit MSB executable, PowerPC or cisco 4500, version 1...
gcc:     ELF 32-bit MSB executable, PowerPC or cisco 4500, version 1...
ld:      ELF 32-bit MSB executable, PowerPC or cisco 4500, version 1...
nm:      ELF 32-bit MSB executable, PowerPC or cisco 4500, version 1...
ranlib:  ELF 32-bit MSB executable, PowerPC or cisco 4500, version 1...
strip:   ELF 32-bit MSB executable, PowerPC or cisco 4500, version 1...
```

我们必须选择适当的目录来存放这些二进制文件，并且为这些重新存放的二进制文件建立符号链接，因为有些 GNU 工具程序（包括 *gcc*）会到固定的 *\$(PREFIX)/\$(TARGET)/bin* 目录寻找其他的 GNU 工具程序，而且如果在那里找不到目标板二进制文件的话，则会使用主机的工具程序。结果自然会编译失败，因为误用了系统工具。编译器具有缺省的搜寻路径可用来寻找二进制文件。我们可以使用如下的编译器选项来检查此路径（请注意，因为版面的关系，过长的每一行会折到下一行）：

```
$ i386-linux-gcc -print-search-dirs

install: /home/karim/control-project/daq-module/tools/lib/gcc-lib/
i386-linux/2.95.3/
programs: /home/karim/control-project/daq-module/tools/lib/gcc-lib/
i386-linux/2.95.3:/home/karim/control-project/daq-module/tools/lib/gcc-lib/
i386-linux/2.95.3:/usr/lib/gcc/i386-linux/2.95.3:/usr/lib/gcc/i386-linux/2.95.3:/home/
karim/control-project/daq-module/tools/i386-linux/bin/i386-linux/2.95.3:/home/
karim/control-project/daq-module/tools/i386-linux/bin/
libraries: /home/karim/control-project/daq-module/tools/lib/gcc-lib/
i386-linux/2.95.3:/usr/lib/gcc/i386-linux/2.95.3:/home/karim/control-project/
daq-module/tools/i386-linux/lib/i386-linux/2.95.3:/home/karim/control-project/
daq-module/tools/i386-linux/lib/
```

programs 那一行的第一个条目就是 `$(PREFIX)/lib/gcc-lib/i386-linux/2.95.3`，此目录包含了 gcc 的链接库与工具程序。若把二进制文件移往此目录，则交叉编译器将会使用它们，而不会使用原生的工具：

```
$ mv as ar gcc ld nm ranlib strip \  
> $(PREFIX)/lib/gcc-lib/i386-linux/2.95.3
```

与此同时，原生的工具链将可以继续正常运行。在应用程序仍然只会到 `$(PREFIX)/$(TARGET)/bin` 目录寻找工具程序的情况下，我们还要为这些重新配置的二进制文件建立符号链接。然而，大多数应用程序都不会只寻找这个目录，所以跳过这个步骤基本上没什么问题。不过，以主机二进制文件的符号链接来取代实际的主机二进制文件的好处是，当你需要将 `$(PREFIX)/$(TARGET)/bin` 目录的内容复制到目标板的根文件系统时，这样可让你比较容易分清楚哪些是目标板二进制文件，哪些不是。如下的命令可以为你建立这些符号链接：

```
$ for file in as ar gcc ld nm ranlib strip  
> do  
> ln -s $(PREFIX)/lib/gcc-lib/i386-linux/2.95.3/$file .  
> done
```

不管你使用的是哪个类型的主机或 gcc 版本，在你建立跨平台开发工具链期间都会产生类似 `$(PREFIX)/lib/gcc-lib/i386-linux/2.95.3` 的目录。正如所见，此目录的路径名称由目标板类型及 gcc 版本组成。因此，对任何人而言，这个目录应该都会放在 `$(PREFIX)/lib/gcc-lib/$(TARGET)/GCC_VERSION`，其中的 `GCC_VERSION` 就是跨平台开发工具链使用的 gcc 版本。

最后，想要节省磁盘空间的话，在工具链的组件安装好之后，或许你可以选择舍弃 `$(PRJROOT)/build-tools` 目录的内容。这种方案相当吸引人，因为“建立目录”现在已经使用了大约 600 MB 的磁盘空间。不过，我劝你先别急着使用 `rm -rf` 命令，最好三思而行。因为你往后可能会遇到意想不到的问题，而需要你再度钻研这个目录里的内容。如果坚持要收回“建立目录”占用的空间，折衷的办法是先等一两个月，看看是否还会用到这些内容再说。

使用工具链

现在你已经拥有了全功能的跨平台开发工具链，对它的使用将会像原生的 GNU 工具链一样频繁，只不过跨平台开发工具链中每个命令的名称都前置了额外的目标板名称。对目标板来说，使用的不是 `gcc` 和 `objdump`，而是 `i386-linux-gcc` 和 `i386-linux-objdump`。

下面这个针对 DAQ 模块的“命令监控程序”设定的 Makefile 文件，便是一个使用跨平台工具链的好例子：

```

# 工具程序的名称
CROSS_COMPILE = ${TARGET}-
AS              = $(CROSS_COMPILE)as
AR              = $(CROSS_COMPILE)ar
CC              = $(CROSS_COMPILE)gcc
CPP             = $(CC) -E
LD              = $(CROSS_COMPILE)ld
NM              = $(CROSS_COMPILE)nm
OBJCOPY         = $(CROSS_COMPILE)objcopy
OBJDUMP         = $(CROSS_COMPILE)objdump
RANLIB          = $(CROSS_COMPILE)ranlib
READELF         = $(CROSS_COMPILE)readelf
SIZE            = $(CROSS_COMPILE)size
STRINGS         = $(CROSS_COMPILE)strings
STRIP           = $(CROSS_COMPILE)strip

export AS AR CC CPP LD NM OBJCOPY OBJDUMP RANLIB READELF SIZE STRINGS \
        STRIP

# 建立时的设定值
CFLAGS          = -O2 -Wall
HEADER_OPS      =
LDFLAGS         =

# 安装时使用的变量
EXEC_NAME       = command-daemon
INSTALL         = install
INSTALL_DIR     = ${PRJROOT}/rootfs/bin

# 建立时需要的文件
OBJS            = daemon.o

# 建立时使用的规则
all: daemon

.c.o:
        $(CC) $(CFLAGS) $(HEADER_OPS) -c $<

daemon: ${OBJS}
        $(CC) -o ${EXEC_NAME} ${OBJS} $(LDFLAGS)

install: daemon
        test -d ${INSTALL_DIR} || ${INSTALL} -d -m 755 ${INSTALL_DIR}
        ${INSTALL} -m 755 ${EXEC_NAME} ${INSTALL_DIR}

clean:
        rm -f *.o ${EXEC_NAME} core

distclean:
        rm -f *~
        rm -f *.o ${EXEC_NAME} core

```

这个 Makefile 的第一个部分指定了将会用来建立目标板应用程序的各个工具链工具程序。每个工具程序的名称都会前置目标板的名称。因此，CC 的值将会是我们前面所建立

的 *i386-linux-gcc*。除了定义工具程序的名称，我们还会导出这些值，好让该 Makefile 后面调用的其他 Makefile 也能使用相同的名称。这样的“建立架构”在（以一个主要目录包含许多子目录的）大型项目中十分常见。

该 Makefile 的第二个部分用来定义建立时的设定值。CFLAGS 提供的标志将会在任何 C 程序文件建立期间使用。如同我们在前一节所见，编译器已经可以使用正确的路径找到目标板的链接库，因此不用设定链接标志变量 LDFLAGS 的值。如果编译器无法指向正确的链接库，或是会用到主机的链接库（如果遵照前面的指示做，应该不会发生此事），我们就必须用链接标志来通知编译器应该使用哪些链接库，方法如下所示：

```
LDFLAGS      = -nostdlib -L${TARGET_PREFIX}/lib
```

如果希望以静态方式链接应用程序，则必须在 LDFLAGS 变量中加入 *-static* 选项。这样产生的执行文件就不会依赖任何共享链接库。但已知标准的 GNU C 链接库相当大，因此会产生出一个非常大的二进制文件。例如，一个只是使用 `printf()` 函数打印出“Hello World!”字符串的简单程序，其二进制文件在动态链接之后会少于 12 KB，但是在静态链接并经 `strip` 处理之后大约有 350 KB。

该 Makefile 的第三个部分（安装时使用的变量）指出了应该将哪个二进制文件以何种方式安装到何处。就此例来说，它将会以 *install* 程序将 `command-daemon` 这个二进制文件安装到目标板的根文件系统中的 */bin* 目录。

就“命令监控程序”这个例子来说，我们只需要建立一个文件。所以在此 Makefile 的第四个部分（建立时所需要的文件）只需要列出这一个文件即可。然而，如果在 LDFLAGS 变量中使用了 *-nostdlib* 选项（通常不需要这么做），那么还需要变更此 Makefile 的第四个部分以及第五个部分（用来产生二进制文件的规则）：

```
STARTUP_FILES = ${TARGET_PREFIX}/lib/crt1.o \
                 ${TARGET_PREFIX}/lib/crti.o \
                 ${PREFIX}/lib/gcc-lib/${TARGET}/2.95.3/crtbegin.o
END_FILES      = ${PREFIX}/lib/gcc-lib/${TARGET}/2.95.3/crtend.o \
                 ${TARGET_PREFIX}/lib/crtn.o
LIBS           = -lc
OBJS           = daemon.o
LINKED_FILES   = ${STARTUP_FILES} ${OBJS} ${LIBS} ${END_FILES}
...
daemon: ${OBJS}
        $(CC) -o ${EXEC_NAME} ${LINKED_FILES} ${LDFLAGS}
```

此处，我们将 *crt1.o*、*crti.o*、*crtbegin.o*、*crtend.o* 和 *crtn.o* 等目标文件和 *daemon.o*（由我们自己的 C 程序文件产生）链接成一个执行文件。前面这 5 个目标文件的作用分别是启动、初始化、构造、析构和结束，它们通常会被自动链接到应用程序中。例如，应用程序的 `main()` 函数就是通过这些文件来调用的。因为在此例中，我们要求编译器不要

进行标准的链接，所以我们必须指明这些必要的目标文件。当你停用标准的链接功能时，倘若未指明这些文件必须链接进来的目标文件，链接器将会抱怨找不到 `_start` 符号，并因而导致链接失败。请注意，将目标文件提供给编译器的次序很重要，因为 GNU 链接器（编译器会自动调用该链接器进行目标文件的链接）只是个单次处理链接器。

此 Makefile 的第五个部分（建立时所使用的规则）与你在标准的、原生的 Makefile 中看到的内容非常像。我加入了 `install` 规则以便让安装过程自动化。你也可以选择不要加入此类规则，但是你必须以手动方式将执行文件复制到正确的目录。

在你当地的目录中有了这个 Makefile 和源文件之后，只须要键入 `make` 就可以为目标板建立程序。如果想在自己的主机上建立原生的执行文件来测试应用程序的话，可以使用如下的命令行：

```
$ make CROSS_COMPILE=""
```

C 链接库的替代品

考虑到嵌入式系统的限制和极限，标准 GNU C 链接库实在太太，不适合应用在目标板上。因此，我们必须找到一个功能完全并且相对较小的 C 链接库。

渐渐地，已经有一些链接库的实现优先考虑到这些问题。接下来，我们将会探讨两种最重要的 C 链接库替代品 `uClibc` 和 `diet libc`。我们会对这两种链接库提供背景信息，以及说明如何为目标板建立链接库，如何使用这些链接库建立应用程序。

uClibc

`uClibc` 链接库源自 `uClinux` 计划。`uClinux` 计划提供的 Linux 版本可以在无 MMU 的处理器上执行。然而，后来链接库变成了独立的计划，并且支持一些具有或不具有 MMU 或 FPU 的处理器。撰写本书时，`uClibc` 支持的处理器架构，第三章都有深入的探讨。在这些架构上，`uClibc` 可以用来作为共享链接库，因为它可以为各种架构提供原生的共享链接库加载器。如果 `uClibc` 并没有为某个架构实现共享链接库加载器，则必须使用 `glibc` 的共享链接库加载器。

有了 `uClibc` 就不需要使用 GNU C 链接库，因为 `uClibc` 提供了类似的功能。当然，`uClibc` 不如 GNU 链接库完整，而且也不准备遵循 GNU 链接库遵循的所有标准。例如，`uClibc` 会省略很少用到的函数及函数特性。然而，使用 GNU C 链接库能够编译成功的大多数应用程序，也可以使用 `uClibc` 来编译和执行。为达此目的，`uClibc` 的开发者优先维护与

C89、C99 和 SUSv3（注 6）的兼容性。他们经常会使用大量的测试套件来确保 uClibc 符合这些标准。

要获得 uClibc 可到该计划位于 <http://uclibc.org/> 的网站下载或使用 CVS 获得 tar-gzip 形式或 tar-bzip2 形式的包。uClibc 发行套件采用的是 LGPL 的版权。如果需要帮助的话，该计划的网站上备有 FAQ 文件，还可以订阅 uClibc 邮件论坛，或是浏览已归档的邮件论坛信件。下一节，我们将会使用 uClibc 0.9.16 进行说明，尽管如此，其内容应该同样适用于后续版本。0.9.16 之前的版本使用的是不同的配置系统，所以其内容不会涵盖在接下来的讨论中。

链接库的设置

这种设置的第一步就是下载 uClibc 包并将源码取出并放到 `${PRJROOT}/build-tools` 目录。与 GNU 工具链组件不同的是，我们使用包自己的目录（而不是使用另一个目录）来进行建立的工作。这主要是因为 uClibc 并不支持在自己目录之外进行建立的工作。然而，其余的建立工作跟其他工具没什么两样，主要的步骤不外是配置、建立以及安装。

从包取出源码后，我们会移往这种设置的 uClibc 目录：

```
$ cd ${PRJROOT}/build-tools/uClibc-0.9.16
```

uClibc 的配置放在名为 `.config` 的文件中，此文件应该放在包的根目录中。为了简化配置工作，uClibc 附带了一个配置系统，该系统会根据我们选择的设定值自动产生 `.config` 文件。它跟我们在第五章将要探讨的内核配置工具程序非常像（注 7）。这个配置系统有各种操作方式，详见包目录包含的 `INSTALL` 文件。设定 uClibc 的最简单方式，就是使用以 `curses` 为基础的终端配置菜单：

```
$ make CROSS=i386-linux- menuconfig
```

这道命令会显示一个菜单，可以使用方向键、Enter 和 Esc 等按键来操作。主菜单中包含了一组次级菜单，这些次级菜单让我们能够设定 uClibc 配置的各个方面。在主菜单层，配置系统允许我们加载和保存配置文件。如果我们在这个菜单项按下了 Esc 键，配置系统将会问我们是否要将配置保存进 `.config` 文件。

在上面的命令中，我们之所以将 `CROSS` 设成 `i386-linux-`，是因为（正如前文所说）我们的跨平台工具的文件名会被前置 `i386-linux-` 这个字符串。接着我们还会编辑 `Rules.mak`

注 6： 即 Single UNIX Specification Version 3。

注 7： uClibc 的配置系统实际上源自 Roman Zippel 的内核配置系统，该系统包含在 Linux 2.5 开发版系列中。

文件，将 CROSS 变量设成 `${TARGET}-`，这样以后要为不同的目标板建立 uClibc 时就不需要设定 CROSS= 了。

你可以在主菜单中看到以下次级菜单：

- Target Architecture Features and Options
- General Library Settings
- Networking Support
- String and Stdio Support
- Library Installation Options
- uClibc hacking options

通过配置系统的次级菜单，我们可以设定许多选项。另外，我们还可以对每个选项使用？键以获得相关的信息。当你对于一个选项按下此键时，配置系统会显示一段如何使用该选项的说明，并提供它的默认值。共有两种选项：一种用来指定建立、安装和操作 uClibc 时需要的工具和目录的路径，另一种用来选择 uClibc 包含的功能。

我们一开始会设定 Target Architecture Features and Options 和 Library Installation Options 次级菜单中的工具和目录路径。表 4-7 列出了在这些次级菜单中我们必须设定的值，让 uClibc 的编译和安装符合我们的工作空间的需要。对每个选项来说，出现在括号中的名称就是 uClibc 配置系统内部使用的变量。想要了解 .config 文件的内容，必须先知道这些变量的用途。

表 4-7: uClibc 工具和目录路径的设定值

选项	设定值
Linux kernel header location (KERNEL_SOURCE)	<code>\${PRJROOT}/kernel/linux-2.4.18</code>
Shared library loader path (SHARED_LIB_LOADER_PATH 译注 a)	<code>/lib</code>
uClibc development environment directory (DEVEL_PREFIX)	<code>\${PRJROOT}/tools/uclibc</code>
uClibc development environment system directory (SYSTEM_DEVEL_PREFIX 译注 a)	<code>\$(DEVEL_PREFIX)</code>
uClibc development environment tool directory (DEVEL_TOOL_PREFIX 译注 b、译注 c)	<code>\$(DEVEL_PREFIX)/usr</code>

译注 a. 对编译本书当时的最新版 uClibc 0.9.26 来说，此变量已更名为 SHARED_LIB_LOADER_PREFIX。

译注 b. 对 uClibc 0.9.26 来说，此变量已不存在。

译注 c. 对 uClibc 0.9.26 来说，新增了一个变量 RUNTIME_PREFIX，其次级菜单名称为 uClibc runtime library directory。

请注意，此处我们使用了 `${PRJROOT}/tools` 而不使用 `${PREFIX}`。尽管在我们的命令脚本中，前者就是我们为 `PREFIX` 环境变量设定的值。这是因为 `uClibc` 在它的 `Makefile` 和相关的命令脚本中也使用了 `PREFIX` 变量，但用法跟我们不一样。主要的差别在于，它使用此变量指向各个安装位置，而我们使用此变量指向主要的安装位置。

`KERNEL_SOURCE` 变量应该指向你将会在目标板上使用的内核源码。如果此值设定不正确，将会导致应用程序无法工作，因为 `uClibc` 不会提供跨内核版本的二进制文件兼容性。

`SHARED_LIB_LOADER_PATH` 就是你将会在目标板上存放共享链接库的目录。所有与 `uClibc` 有链接关系的二进制文件都会将此值硬编码。如果后来改变了共享链接库的位置，则必须重建 `uClibc`。此处之所以将目录设成 `/lib`，是因为共享链接库通常会放在这个位置。

`DEVEL_PREFIX` 就是 `uClibc` 将会被安装的目录。如同其他的工具，我们将会把它放到 `${PRJROOT}/tools` 目录之下。`SYSTEM_DEVEL_PREFIX` 和 `DEVEL_TOOL_PREFIX` 是另两个安装变量，用来控制某些 `uClibc` 二进制文件的安装，这通常只对想要建立 `RPM` 或 `dpkg` 套件的人有用。以我们的设置来说，我们可以将 `SYSTEM_DEVEL_PREFIX` 的值设成跟 `DEVEL_PREFIX` 一样，以及将 `DEVEL_TOOL_PREFIX` 设成 `${DEVEL_PREFIX}/usr`。结果，所有前置目标板名称的 `uClibc` 二进制文件，例如 `i386-uclibc-gcc`，会被安装到 `${PRJROOT}/tools/uclibc/bin`；所有未前置目标板名称的 `uClibc` 二进制文件，例如 `gcc`，则会被安装到 `${PRJROOT}/tools/uclibc/usr/bin`。正如稍后所见，要使用 `uClibc`，只需要将 `${PRJROOT}/tools/uclibc/bin` 加入搜索路径就可以了。

现在让我们检查每个配置次级菜单中可以看到选项。正如我稍早所说，可以用 `?` 键从配置系统获得每个选项的进一步信息。因为有些选项跟其他选项的设定值间有依存关系，以下列出的选项，有些可能不会显示在配置系统上。尽管大部分都是开关形式的选项（不是启用就是停用），还是有些选项属于字符串字段，例如我们前面所提到的路径名称，此时必须填入字符串。

Target Architecture Features and Options 次级菜单包含以下选项：

- Target Processor Type
- Target CPU has a memory management unit (MMU) (`UCLIBC_HAS_MMU`)
- Enable floating point number support (`UCLIBC_HAS_FLOATS`)
- Target CPU has a floating point unit (FPU) (`HAS_FPU`)
- Enable full C99 math library support (`DO_C99_MAT`)

- **Compiler Warnings (WARNINGS)**。这是个字符串字段，允许你为编译器设定用于报告警告信息的标志。
- **Linux kernel header location (KERNEL_SOURCE)**。这是前面提到的内核路径。

General Library Settings 次级菜单包含以下选项：

- **Generate Position Independent Code (PIC)(DOPIC)**
- **Enable support for shared libraries (HAVE_SHARED)**
- **Compile native shared library loader (BUILD_UCLIBC_LDSO)**
- **Native shared library loader 'ldd' support (LDSO_LDD_SUPPORT)**
- **POSIX Threading Support (UCLIBC_HAS_THREADS)**
- **Large File Support (UCLIBC_HAS_LFS)**
- **Malloc Implementation**。这个次级菜单允许我们选择两种 malloc 实现 (malloc 和 malloc-930716) 其中之一。
- **Shadow Password Support (HAS_SHADOW)**
- **Regular Expression Support (UCLIBC_HAS_REGEX)**
- **Supports only Unix 98 PTYs (UNIXPTY_ONLY)**
- **Assume that /dev/pts is a devpts or devfs filesystem (ASSUME_DEVPTS)**

Networking Support 次级菜单包含以下选项：

- **IP Version 6 Support (UCLIBC_HAS_IPV6)**
- **Remote Procedure Call (RPC) support (UCLIBC_HAS_RPC)**
- **Full RPC support (UCLIBC_HAS_FULL_RPC)**

String and Stdio support 次级菜单包含以下选项：

- **Wide Character Support (UCLIBC_HAS_WCHAR)**
- **Locale Support (UCLIBC_HAS_LOCALE)**
- **Use the old vfprintf implementation (USE_OLD_VFPRINTF)**

在这一节前面我们已经说明过 Library Installation Options 次级菜单中的选项。不过，为了完整起见，还是将它们列出来：

- **Shared library loader path (SHARED_LIB_LOADER_PATH)**

- uClibc development environment directory (DEVEL_PREFIX)
- uClibc development environment system directory (SYSTEM_DEVEL_PREFIX)
- uClibc development environment tool directory (DEVEL_TOOL_PREFIX)

尽管通常你应该不需要进入 uClibc hacking options 次级菜单，但为了完整起见，还是将它们列出来：

- Build uClibc with debugging symbols (DODEBUG)
- Build uClibc with runtime assertion testing (DOASSERTS)
- Build the shared library loader with debugging support (SUPPORT_LD_DEBUG)
- Build the shared library loader with early debugging support (SUPPORT_LD_DEBUG_EARLY)

以我的 DAQ 模块来说，我会保留这些选项的缺省值。以大多数的目标板来说，我们应该也不用改变这些选项。不要忘了，只要将 *.config* 文件从 uClibc 的目录删除，就可以恢复到缺省值。

uClibc 设定好之后，接着进行编译：

```
$ make CROSS=i386-linux-
```

以我们的设置来说，编译过程大约要耗费 10 分钟的时间。如同 GNU 工具链，可能会在建立期间看到一些可以忽略的警告信息。

建立完成之后，接着安装 uClibc：

```
$ make CROSS=i386-linux- PREFIX="" install
```

以我们在前面所设定的值来说，这将会把所有的 uClibc 组件安装到 *\$(PRJROOT)/tools/uclibc* 目录。如果我们之前已经安装过 uClibc，当安装程序要将文件复制到 *\$(PRJROOT)/tools/uclibc* 目录时，整个过程将会失败。如果有这种情况发生，应该在执行 *make install* 命令之前，先清除 *\$(PRJROOT)/tools/uclibc* 目录中的内容。

用法

我们现在已经准备好，可以将应用程序链接到 uClibc 链接库。为了协助此链接，uClibc 在 *\$(PRJROOT)/tools/uclibc/bin* 目录中安装了几个工具程序。uClibc 安装的主要是编译器和链接器的替代品：*i386-uclibc-gcc* 和 *i386-uclibc-ld*。uClibc 安装的工具程序和符号链接，其名称并非前置 *i386-linux-* 而是 *i386-uclibc-*。uClibc 的编译器和链接器其实只是

包装程序，它们最后还是会调用我们前面建立的 GNU 工具程序，不过它们能够让应用程序用 uClibc 顺利完成建立与链接的工作。

使用这些工具程序的第一步就是修改我们的搜索路径：

```
$ export PATH=${PREFIX}/uclibc/bin:${PATH}
```

你还可能需要变更开发环境命令脚本，让这个路径改变动作自动化。因此，我们为 *develdaq* 新增了一行如下的内容：

```
export PATH=${PREFIX}/bin:${PREFIX}/uclibc/bin:${PATH}
```

使用与稍早相同的 Makefile，我们可以用如下的方式来编译“命令监控程序”：

```
$ make CROSS_COMPILE=i386-uclibc-
```

因为 uClibc 在 x86 架构上预设为共享链接库，所以会产生动态链接的二进制文件。然而，我们仍然可以用静态链接的方式来编译应用程序：

```
$ make CROSS_COMPILE=i386-uclibc- LDFLAGS="-static"
```

稍早所提到的“Hello World!”程序，如果动态链接共享的 uClibc 链接库，其大小只有 2 KB；若采用静态链接的方式，则有 18 KB。这就是相同的程序链接 uClibc 与链接 glibc 最大的不同之处。

diet libc

diet libc 计划的发起者是 Felix von Leitner，此计划目前仍是由 Felix von Leitner 负责维护，其目标与 uClibc 类似。然而，与 uClibc 不同的是，diet libc 的发展并非建立在既有链接库的成果上，而是依据二进制文件最小化和效率最佳化的理念从头开始撰写而成。因此，不管是就二进制文件的大小或是就执行速度来看，diet libc 都比 glibc 优越许多。但是，与 uClibc 相比，我并未发现任何显著的不同。

第三章提到的处理器架构中，diet libc 只支持 ARM、MIPS、x86 和 PPC。此外，diet libc 的作者们偏爱静态链接的方式。所以，尽管在某些平台上 diet libc 可当作共享链接库来使用，不过缺省情况下还是会当成静态链接库来用。

评估是否使用 diet libc 的时候，请留意最重要的一个问题，那就是它的许可方式。大多数的链接库，包括 uClibc 在内，通常会采用 LGPL 的许可方式，而 diet libc 采用的是 GPL 的许可方式。正如我在第一章所做的说明，这代表如果程序代码与 diet libc 链接，则产生的二进制文件就变成了 diet libc 的衍生作品，因此二进制文件若不遵照 GPL 的规定就无法对外发行。如果希望以非 GPL 的方式发行跟 diet libc 链接的二进制文件，则你可以

从该套件的主要作者处获得商业许可（注8）。然而，如果不想处理许可方面的问题，可能会宁可使用 uClibc。

欲获得 diet libc 可到该计划位于 <http://www.fefe.de/dietlibc/>（注9）的网站下载 tar-bzip2 形式的压缩包，或是以 CVS 取回源码。该包附带 FAQ 文件和安装说明。接下来，我将会以 diet libc 0.21 为例进行说明，不过我的说明应该同样适用于之前和之后的版本。

链接库的设置

如同 uClibc，设置 diet libc 的第一步就是将它下载到我们的 `${PRJROOT}/build-tools` 目录。同样地，我们将会在该套件的源码所在的目录（而不是像 GNU 工具链那样在另一个目录）建立链接库。而且不需要对 diet libc 进行配置，我们可以立即进入建立阶段。

一旦从套件取出源码之后，接着移往 diet lib 的源码目录进行设置的工作：

```
$ cd ${PRJROOT}/build-tools/dietlibc-0.21
```

在为目标板建立套件之前，我们会先为主机建立套件。要为目标板建立 diet libc，以及使用这个 diet libc 来建立应用程序，必须先产生 *diet* 工具程序：

```
$ make
```

在我们的设置中，这会为 PPC 架构产生 diet libc 并放到 *bin-ppc* 目录。我们接着可以为目标板编译 diet libc：

```
$ make ARCH=i386 CROSS=i386-linux-
```

你会看到比其他套件还要多的警告信息，不过你可以忽略不管。这里必须告诉 Makefile，我们要为哪种架构建立 diet libc，以及跨平台开发工具文件名的前置字符串。

套件建立好之后，接着进行安装：

```
$ make ARCH=i386 DESTDIR=${PREFIX}/dietlibc prefix="" install
```

这会将 diet libc 组件安装到 `{PREFIX}/dietlibc` 目录。当我们为目标板建立套件时，同样必须告诉 Makefile 要为哪种架构建立 diet libc。我们必须使用 DESTDIR 变量来指定安装目录，并且重新设定 Makefile 的内部变量 prefix（请注意，此变量不同于大写的环境变量 PREFIX）。

注8： 开发者向主要作者所获得的商业授权，不知是否包含其他作者在内。

注9： 注意最后一个 “/”。如果省略此斜线符号，该网站将会找不到网页。

将diet libc 安装到适当的目录之后，我们需要对此安装做一下修正。x86 版本的diet libc 安装之后，我们也将x86版本的diet 工具程序安装到了`${PREFIX}/dietlibc/bin`目录。因为我们想在主机上编译应用程序，所以必须用前面建立的原生diet 工具程序覆盖掉它：

```
$ cp bin-ppc/diet ${PREFIX}/dietlibc/bin
```

用法

如同uClibc，使用diet libc 的方法包括修改搜索路径，以及使用diet libc 提供的打包程序（diet）来链接应用程序。然而，与uClibc 不同的是，我们不必以链接库特有的工具来取代交叉开发工具，我们只需要将diet libc 打包程序放在所调用的工具之前即可。

首先，我们必须变更搜寻路径，让它包含存放diet libc 二进制文件的目录：

```
$ export PATH=${PREFIX}/dietlibc/bin:${PATH}
```

同样地，还可能需要变更开发环境命令脚本。例如，可以将`develdaq` 命令脚本中的搜索路径改成这个样子：

```
export PATH=${PREFIX}/bin:${PREFIX}/dietlibc/bin:${PATH}
```

请注意，此处假定你不会同时使用uClibc 和diet libc。因此，搜索路径中只加入了diet libc。如果想在系统开发期间同时使用diet libc 和uClibc，那么必须同时加入它们的路径。

以diet libc 编译“控制后台程序”时，我们会使用如下的命令行：

```
$ make CROSS_COMPILE="diet i386-linux-"
```

因为diet libc 主要是静态链接库，会产生静态链接的二进制文件，所以你不需要为命令行加上`LDFLAGS="-static"`。前文提到的“Hello World!”程序，若与diet libc 链接将会产生24 KB 的二进制文件。

Java

自从Sun 于1995 年推出Java™ 以来，它就成为了世界上最重要的语言之一。今日，各种的计算机化系统中，包括嵌入式系统，都可以发现它的踪迹。尽管目前在嵌入式程序设计领域里Java 不如C 普遍，但是有逐渐增加的趋势。

此处将不会深入探讨Java 或与它有关的任何技术。这方面的书籍已经太多了，包括O'Reilly 出版的许多相关书籍。然而，在继续下去之前，我们必须先检查一个基本问题。实质上，与Java 有关的任何讨论都会涵盖三个方面的内容：Java 程序设计语言、Java 虚拟机以及Java 运行时环境，Java 运行时环境由各种Java 类组成。

有许多（自由的和私有的）套件可以在 Linux 中提供 Java 的功能。我们的讨论将只会专注在自由的套件上。尤其是，我们将会探讨 Blackdown 计划的开放源码虚拟机，以及用来编译 Java 程序语言的 GNU 编译器。但是，我将不会说明这些工具的安装与用法，因为它们在 Linux 工作站上与嵌入式 Linux 系统中的安装与用法会有些不同。然而，我将告诉你何处可以找到相关的文件。

Blackdown 计划

Blackdown 计划 (<http://www.blackdown.org/>) 是人们将 Sun 的 Java 工具移植到 Linux 的结果。此计划完全建立在 Sun 的 Java 源码基础上，目的是把 Sun 的 Java 工具移植到 Linux，其成果包括供 Linux 工作站和服务端使用的 Java Development Kit (Java 开发工具箱，JDK) 和 JRE。

此计划虽享有特权，不过它与 Sun 的关系有时让它很为难。因为这计划完全建立在 Sun 的源码基础上，而且此源码并未采用开放源码的许可方式（注 10），这完全是出于 Sun 为了协助 Linux 社群的善意。

实际上，Blackdown 计划并未发行任何源码。它只发行为各种处理器架构预先建立好的二进制文件，这些二进制文件就是开发者移植 Sun 的 Java 工具的成果。因为该计划的 FAQ 指出，要获得源码必须跟 Sun 接洽。

Sun 和 Blackdown 之间的许可条款指出，允许用户下载 JDK 供自己使用，但不经 Sun 的同意不得散布。然而，用户可以下载并散布 JRE，因此 JRE 的限制较少。

在发布新的版本之前，Blackdown 团队的移植成果必须通过 Sun 的兼容性测试。因此，之前版本所支持的架构，后续版本未必会支持。例如，1.3.0-FCS 同时支持 PPC 和 x86，但 1.3.1-rc1 只支持 ARM。欲了解 Blackdown 发布过哪些版本及其支持的平台，可参考 <http://www.blackdown.org/java-linux/ports.html>。

欲执行 JDK 或 JRE，至少需要 glibc，如果想要使用 AWT 类，则还需要 X Window System 与它的链接库。不过由于大多数嵌入式系统的资源有限，除非具备大型的存储设备以及足够的处理能力，否则无法执行这类应用程序。

Blackdown 计划的进一步信息，包括它所提供的工具、如何安装这些工具、如何操作这

注 10： 你可以在 Sun Community Source License (SCSL) 的授权下获得 Sun 的 Java 工具程序的源码。SCSL 并非 Open Source Initiative (OSI) 认可的许可证。想了解 OSI 认可过哪些许可证，可到 <http://opensource.org/licenses/> 查看完整的清单。

些工具以及许可问题，可参考 Blackdown FAQ (<http://www.blackdown.org/java-linux/docs/support/faq-release/>)

开放源码虚拟机

由于 Blackdown 受制于 Sun，为了不使用任何 Sun 源码的情况下提供开放源码和全功能的 JVM，于是出现了一些开发计划。其中最值得注意的就是 Kaffe。

因为在嵌入式 Linux 计划中，JVM 的用法将因开放源码虚拟机而异，所以我将只会简单介绍一下这些虚拟机，并不会提供与使用方法有关的任何信息。建议浏览每个虚拟机计划的网站，以便了解每个开发团队目前的进展情况。

Kaffe JVM (<http://www.kaffe.org/>) 是根据 Transvirtual Inc. 销售的产品 KaffePro VM，以 clean-room 方式实现的(注 11)。尽管该计划提供的虚拟机并没有和 Sun 的虚拟机 100% 兼容，不过该计划的网站宣称，它仍然是 Sun 虚拟机主要的开放源码替代品。

此外，有些计划往后可能会变得更加重要，例如 Japhar (<http://www.japhar.org/>)、Kissme (<http://kissme.sourceforge.net/>)、Aegis (<http://aegisvm.sourceforge.net/>) 和 Sable VM (<http://www.sablevm.org/>)。开放源码虚拟机计划的完整清单，可参考 joeq VM (<http://joeq.sourceforge.net/>) 于 http://joeq.sourceforge.net/other_os_java.htm 所提供的“open source VM project”列表。每个计划的网站上都提供有如何安装和操作其虚拟机的信息。

GNU Java 编译器

GNU Compiler for the Java programming language (gcj) 是 GNU 计划的一部分，它是 gcc 的一个延伸，可用来处理 Java 源码和 Java bytecode。尤其是，gcj 可以将 Java 源码或 Java bytecode 编译成原生的机器码。此外，它通常会被称为“预先”(ahead-of-time, AOT) 编译器，因为它可以将 Java 源码直接编译成原生码，然而一般的“即时”(just-in-time, JIT) 编译器却是在运行时将 Java bytecode 转换成原生码。而且 gcj 还具备与 JDK 的 *java* 命令等效的 Java 解释器。

GNU gcj 是个相当活跃的计划，而且 Java 的内核类链接库大都已经成为 gcj 运行时链接库的一部分。尽管大多数的窗口组件(例如 AWT)都仍在开发中，不过它目前的编译器和运行时环境，已经能够用来编译和执行大多数的命令行应用程序。

注 11: 也就是说，它是从头开始撰写而成的，并未用到 Sun 的任何源码。

如同其他的 GNU 计划，gcj 的文档也相当好。该计划位于 <http://gcc.gnu.org/java/> 的网站是一个不错的起点。你可以在该网站上找到安装指南、FAQ，以及使用 *gdb* 进行 Java 应用程序调试的指南。你应该根据这个安装指南和稍早所提到的建立 GNU 工具链的指南来为目标板建立 gcj。

Perl

自从 Larry Wall 于 1987 年推出 Perl 程序语言以来，这个程序语言就拥有了自己的一片天空。如果对 Perl 有兴趣，可以阅读 Wall、Christiansen 和 Orwant 合著的《Programming Perl》或是 Schwartz 所著的《Learning Perl》（这两本书都出版自 O'Reilly）。简单地说，Perl 是一种整合性的语言，只要你同意 Perl Artistic License 和 GNU GPL 的许可条款就可以到位于 <http://www.cpan.org/> 的 Comprehensive Perl Archive Network（Perl 综合典藏网，CPAN）获得它的编译器、工具以及链接库的源码。因为只存在一种 Perl 工具集，所以你将不需要为了找到最佳套件而评估不同的工具集。

要让 Perl 程序在目标板上执行，必须为目标板正确地编译 Perl 解释器。遗憾的是撰写本书时 Perl 还无法交叉编译成功。然而，解决此问题的努力仍持续进行中。根据 Jarkko Hietaniemi（5.8 版的负责人）的说法，Perl 5.8.0 本身应该能够交叉编译成功。在当时，5.7 开发分支对于整个 Perl 套件的缩小版提供有两种选项：microperl 和 miniperl。请注意，这两种选项都是相同套件的一部分，所以只需要使用同一个套件。

microperl

microperl 建立选项是 Simon Cozens 根据 Ilya Zakhareivh 的构想实现而成的。microperl 绝对是你建立出来的 Perl 解释器中最小的一个，除了 ANSI C 和 *make* 工具程序，它与外界没有任何关系。与其他的建立方式不同，microperl 并不需要你执行配置命令脚本——这个命令脚本在为套件的建立产生适当的文件之前，会在欲安装的机器上进行大规模的测试。缺省的配置文件提供的最起码设定值已经能够让内核的 Perl 解释器顺利完成建立的工作。然而，该解释器并不会因此而遗漏 Perl 语言的内核特性。当然，该解释器将无法支持完整解释器的所有特性，不过执行基本的 Perl 应用程序倒是没什么问题。因为它的程序代码此刻被认为是“实验性的”，所以你必须自己评估是否要使用 microperl。

我曾经使用前面设置的工具链、uClibc 和 Perl 5.7.3 为 DAQ 模块成功地建立出 microperl。所产生的解释器能够正确地执行所有无外界引用的 Perl 程序。然而，所执行的程序如果用到任何标准的 Perl 模块就会失败。

要为目标板建立 `microperl`，首先必须从 CPAN 下载某个 Perl 版本，并将源码取出放到 `${PRJROOT}/sysapps` 目录。之所以将套件放到 `sysapps` 目录，是因为它只会在目标板上执行，而且无法拿来为目标板建立其他软件套件。从套件取出源码后，我们将会移往源码所在目录，以便进行建立的工作。此处，我们将不会像 GNU 工具链那样使用不同的建立目录，因为 Perl 并不支持这种建立方式。

```
$ cd ${PRJROOT}/sysapps/perl-5.7.3
```

因为 `microperl` 是 Perl 的缩小版，我们并不需要设定任何配置。只要使用正确的 Makefile，以及指示它使用 `uClibc` 编译器打包程序（而不要使用标准的 `gcc` 编译器）就可以完成套件的建立工作：

```
$ make -f Makefile.micro CC=i386-uclibc-gcc
```

这将会在套件的根目录中产生 `microperl` 二进制文件。此二进制文件并不需要任何其他的 Perl 组件，可以直接复制到目标板的根文件系统（`${PRJROOT}/rootfs`）中的 `/bin` 目录。

如果与 `glibc` 或 `uClibc` 动态链接并经过 `strip` 处理，则 `microperl` 二进制文件的大小约有 900 KB。当采用静态链接的方式并经过 `strip` 处理时，如果链接的是 `glibc`，则二进制文件的大小有 1.2 MB；如果链接的是 `uClibc`，则二进制文件的大小有 930 KB。正如所见，若考虑二进制文件尺寸的因素，`uClibc` 确实是比较好的选择。

若想获得如何建立 `microperl` 的进一步信息，可检查 `Makefile.micro` 这个 Makefile，以及 `uconfig.sh` 这个命令脚本。因为 `microperl` 的发展仍在进行中，预期会有更多的文件问世。

miniperl

`miniperl` 比 `microperl` 复杂一点，它能够提供标准 Perl 解释器大部分的功能。`miniperl` 所欠缺的主要组件是 `DynaLoader XS` 模块，该模块让 Perl 子进程能够调用 C 函数。因此 `miniperl` 无法动态加载 `XS` 模块。然而，这对将会执行 `miniperl` 的系统类型来说，并不会构成太大的问题。

如同建立标准的 Perl 解释器，`miniperl` 也需要你执行 `Configure` 命令脚本，以便判断系统具备哪些能力。因为必须为目标板建立 Perl 解释器，所以命令脚本会要求你提供相关的信息，以便与目标板进行联系。这些信息包括远程主机名称、远程用户名称、目标板上的目录。然后命令脚本会使用这些信息在目标板上执行它的测试程序，以便产生正确的建立文件。

请注意，建立 `miniperl` 时，主机和目标板之间必须存在直接的网络链接。基本上，如果目标板并不具备某种形式的联网能力，将无法为它建立 `miniperl`。

此处并不会提供建立及安装 miniperl 的细节，因为在 Perl 5.7.3 套件附带的 *INSTALL* 文件中，已经在标题为“Cross-compilation”的那个部分说明的很清楚了。

不能进行交叉编译

正如所见，并非所有的 Perl 套件都可以轻易完成交叉编译。事实上，有很多 Perl 套件在设计上并未考虑交叉编译。本书也只能略举一二，但无法全部提到。

除了修改建立命令脚本及使用编译技巧迫使套件为另一个平台进行编译之外，有时惟一可行的方法，就是在套件所要执行的目标板上进行实际建立的工作。但是，这方法对大多数的嵌入式系统来说似乎有点不切实际，因为这类系统的存储空间通常很有限。然而，第九章提到过，我们可以通过 NFS 来安装系统的根文件系统。只要使用经 NFS 安装的根文件系统，目标板所能存取的存储空间就仅受限于 NFS 服务器所能提供的容量。

在这样的设置中，可以先在主机上为目标板进行 gcc 编译器的交叉编译，然后使用这个编译器以原生的方式在目标板上为任何套件直接进行编译，让套件的建立命令脚本以其预期的方式进行操作。一旦套件完成编译动作，可将其产生的二进制文件和链接库复制到为目标板的内部存储设备定制的小型根文件系统，然后如同任何其他的目标板应用程序一样，实地拿来使用。显然，经交叉编译的 gcc 并不需要与系统放在一起。

Python

自从 Guido van Rossum 于 1991 推出 Python 以来，它便聚集了许多追随者，如同 Perl 一样，Python 也有自己的一片天空。如果对 Python 有兴趣，可以阅读 Mark Lutz 所著的《Programming Python》，或是 Lutz、Ascher 和 Willison 合著的《Learning Python》（这两本书皆出版自 O'Reilly）。Python 常被拿来跟 Perl 比较，所以它通常被应用在与 Perl 相同的用途上，但这已经涉及谁优谁劣的圣战问题，因此我不便多说什么。欲获得 Python 的进一步信息，可浏览位于 <http://www.python.org/> 的主要 Python 网站。只要你接受称为 Python license 的复合式许可证（这是一个经过认可的开放源码许可证），就可以到该网站获得 Python 套件（包括 Python 解释器和 Python 链接库）。

如同 Perl，若想在目标板上执行 Python 程序代码，将需要对解释器进行适当设定。主要的 Python 发行套件并不支持交叉编译，Klaus Reimer 为解决此问题开发出了一个补丁，可以从 <http://www.ailis.de/~k/patches/python-cross-compile.diff> 获得。Klaus 还提供

了一份写得非常好的“Python cross-compiling HOWTO”文件，可以从<http://www.ailis.de/~k/knowledge/crosscompiling/python.php> 获得。

你可以照着 Klaus 的指示，为目标板建立 Python，不过你得将指示中使用的 arm-linux 替换成与目标板相符的名称。你可以根据我们前面建立的项目工具工作空间，下载 Python 套件并将源码取出放到 `${PRJROOT}/sysapps` 目录，就像建立 GNU 工具那样，可以使用 `build-python` 目录进行建立的工作，因为 Python 支持这种建立方式。此外，还需要以 `--prefix=${PREFIX}/${TARGET}/usr` 来取代 HOWTO 文件提供的值。因此，建立好的 Python 最后会被安装到 `${PREFIX}/${TARGET}/usr` 目录。然后你可以将此目录裁剪并复制到目标板的根文件系统。

我对 Python 套件的建立有两点说明。首先，无法用 diet libc 来建立 Python。要建立 Python 必须使用 glibc 或 uClibc。这表示 glibc 或 uClibc 必须放在目标板的根文件系统上。当目标板上的存储空间有限时，建议使用 uClibc。此外，如果想用 uClibc 来建立 Python 的话，必须使用 Manuel Novoa 于 August 27, 2002 在 uClibc 邮件论坛回复 uClibc 0.9.15 发行公告时提供的补丁（译注 1）。

其次，Python 在 `${PREFIX}/${TARGET}/usr/lib/python2.2` 目录中安装了许多链接库，其中包含许多大型文件。你可能需要删除其中不可能用到的组件，以便调整此目录的内容。就其本身来说，经动态链接及 strip 处理的 Python 解释器大小有 725 KB。

然而，Python 的大小以及依存关系并未终止开发者使用它的决心。例如，开发 iPAQ 的 Familiar 发行套件的团队，就将 Python 纳入他们的标准套件清单中。

最后，诚如 Klaus 所说，可能会在建立期间看到一些警告和失败信息。这是因为目标板上遗漏了一些链接库和应用程序。例如，使用 `libtk.a` 和 `libtcl.a` 链接库的 Tkinter 接口将会建立失败，除非你之前曾为目标板交叉编译并安装过 Tcl/Tk。但这并不代表 Python 建立失败。这只代表某个 Python 组件未建立成功。你仍然可以安装并使用为目标板建立的 Python 解释器和模块。

Ada

Ada 是个受到美国国防部（DoD）资助的程序语言。20 世纪 70 年代，DoD 认识到手头有庞大的软件维护问题。因此着手发展新的程序语言，以解决其对程序代码的可维护性和可靠性的迫切需求。ANSI 在 1983 年首次通过 Ada 标准，并于 1995 年发表 Ada95 标准。

译注 1: <http://codepoet.org/lists/uclibc/2002-August/004254.html>。

后来纽约大学着手开发以 gcc 为基础的 Ada 编译器，并开发出了 gnat 这个 GNU Ada 编译器（注 12）。gnat 的开发工作在 Ada Core Technologies Inc. (ACT) 继续进行着，在 gnat 完全整合进主要的 gcc 源码树之前，由 ACT 负责维护 gnat。ACT 不时地以 GPL 版权发布其新近的工作成果，并为某些平台提供预先建立好的二进制文件，这可以从 <ftp://cs.nyu.edu/pub/gnat/> 获得。写作本书时发行的最后一个版本为 gnat 3.14p，这个版本的建立需要 gcc 2.8.1。精确地说，gnat 源码所提供的修正必须加入 gcc 源码，而且必须将 ada 目录复制到 gcc 的源码目录。

遗憾的是，这导致了许多问题。例如，gcc 2.8.1 已经相当旧，而且新近的 gcc 版本大都无法正确建立 gnat。因此，如果要使用 gnat 3.14p，首先必须在系统上安装一个旧的编译器，然后用它来建立 gnat。显然不会有人喜欢这么做。

最近，ACT 的工作被整合到 gcc CVS，而且成为 gcc 3.2 的一部分。尽管你仍然需要 gnat 二进制文件来建立 Ada 编译器，但是未来当 gnat 整合进主流的 gcc 之后，将有可能简化 Ada 在嵌入式 Linux 系统中的使用。

如果对 Linux 上的 Ada 程序设计有兴趣，除了仍在进行的 gnat 计划，还可以找到两个计划。首先是 Ken Burtch 发起的《The Big Online Book of Linux Ada Programming》计划，其目的在于为 Linux 上的 Ada 程序设计提供完整的在线参考手册。你可以到 <http://www.pegasoft.ca/homes/book.html> 以及镜像网站获得该手册。

其次是 *Ada for GNU/Linux Team* (ALT) 在 <http://www.gnuada.org/alt.html> 提供的非 ACT 二进制套件、RPM 和补丁。其中还包含了用来提供 Ada 接口的套件以及链接库，例如 GTK、XML 和 X11。

其他程序语言

当然，Linux 上还支持许多其他的程序语言。不论你要找的是 Forth、Lisp 或 FORTRAN，都可以轻易地在网络上你常用的搜寻引擎中马上找到结果。《Running Linux》（由 O'Reilly 出版）一书的第 13.5 节“Other Languages”是个不错的起点。

各种语言工具的交叉编译和交叉开发能力，要按照工具对工具的方式来评估，因为这些编译器和解释器多半对跨平台开发没有什么帮助。

注 12： 值得注意的是，gnat 完全是用 Ada 写成的。

30 年前的重要软件……

DoD 的软件问题让我回想起两年前所参加的 2000 Usenix 的技术研讨年会。这次是第 25 届技术研讨年会，与会者包括 Dennis Ritchie 和 Ken Thompson，实在很难得。

Evi Nemeth 在一开始的议程解说会上提到 Dennis Ritchie 和 Ken Thompson 与会的事情，并指出他们的成果有持久的价值。她在说明中特别强调 Dennis Ritchie 和 Ken Thompson 的构想会非常长久，并说：“Unix 已经有 30 年的历史了。你知道有哪些软件已经被使用了 30 年？”

此刻，人群中有人突如其来地回答了这个不需回答的反问句：“女士，我在空军任职……”

集成开发环境

Linux 拥有许多集成开发环境（integrated development environment，IDE）。这些 IDE 多半被用来开发原生的应用程序。然而，我们可以借着在 IDE 配置中设定适当的编译器名称来进行交叉开发的定制化。表 4-8 列示了一份开放源码 IDE 清单，其中包含它们的位置，以及它们所支持的嵌入式 Linux 相关程序语言。

表 4-8：开放源码 IDE

IDE	位置	所支持的语言
Anjuta	http://anjuta.sourceforge.net/	Ada、bash、C、C++、Java、make、Perl、Python
Eclipse	http://www.eclipse.org/	C、C++、Java
Glimmer	http://glimmer.sourceforge.net/	Ada、bash、C、C++、Java、make、Perl、Python、x86 assembly
KDevelop	http://www.kdevelop.org/	C、C++、Java
SourceNavigator	http://sources.redhat.com/sourcenav/	C、C++、Java、Python

我并不愿意推荐任何特定的 IDE，因为 IDE 的选用取决于个人的喜好。我自己喜欢使用 *XEmacs* 以及任何 IDE 的命令行。不过，有的人则仍旧喜欢简单易用的 *vi*。你可能会想要看一下各计划所提供的屏幕捉图，以作为最初评定的依据。然而，最后要下决定之前，可能会希望下载 IDE 试用看看。

就普及率来说，KDevelop 或许是此清单中普及率最高的 IDE。尽管 KDevelop 主要的用途是用户应用程序的原生开发，不过我们可以对它进行交叉开发的定制化。Anjuta 是个非常活跃的计划，它的接口与许多流行的 Windows IDE 很类似。SourceNavigator 是 Red Hat 以 GPL 许可条款发行的 IDE，它原先是 Red Hat 的 *GNUPro* 产品的一部分。Glimmer 是以 Gnom 为基础的 IDE，它具备与其他 IDE 类似的能力。Eclipse 是个野心勃勃的计划，其目标是设计出一个可轻易使用插件来扩充的 IDE 骨架。Eclipse 背后受到许多公司的支持，包括 IBM、HP、Red Hat 和 SuSE。

欲获得这些计划的进一步信息，可浏览它们的网站以及网站上所提供的文件。

终端仿真程序

与嵌入式系统的联系，最常见的方式就是在主机端使用终端仿真程序，通过 RS232 串行端口与目标板通信。尽管可供 Linux 使用的终端仿真程序为数不多，但是并非所有的终端仿真程序都适合所有人使用。例如，minicom 与 U-Boot 之间通过串行端口传送文件就有问题。因此我建议各位，与目标板的联系要多试几个终端仿真程序。就算都没有问题，也可能会因而找到最合适的程序。此外，请检查引导加载程序的文件是否有提到与任何终端仿真程序相关的警告信息。

在 Linux 上有三种常用的终端仿真程序：*minicom*、*cu* 和 *kermit*。接下来的各小节将会说明这些工具的设置与配置，但并不包含它们的用法。用法的部分往后可以参阅这些工具的说明文件。

存取串行端口

在使用任何终端仿真程序之前，必须先确定你对主机上的串行端口具有适当的使用权。尤其是，需要对串行端口装置，通常是 */dev/ttyS0*，具有读和写的使用权，以及对 */var/lock* 目录具有读和写的使用权。想要与串行端口对话，必须能够存取 */dev/ttyS0*。想要锁住串行端口的使用权，必须能够存取 */var/lock*。如果不具备这些权利，任何你所使用的终端仿真程序都将会在启动时送出抱怨信息（注 13）。

/dev/ttyS0 的权限位和群组设定值因发行套件而异，有时相同发行套件的不同版本也会有所差异。例如，在 Red Hat 6.2 中，只有 root 用户对它具有读和写的使用权：

```
$ ls -al /dev/ttyS0
crw----- 1 root    tty      4,  64 May  5 1998 /dev/ttyS0
```

注 13： 这一节的说明，可能对你所使用的发行套件不适用。有疑虑时，请参考发行套件提供的说明文件。

如同 `/dev/ttyS0`、`/var/lock` 的权限位和群组设定值也深受发行套件的影响。同样以 Red Hat 6.2 为例，包括 root 用户和 uucp 使用群组的任何成员都可以存取 `/var/lock`：

```
$ ls -ld /var/lock
drwxrwxr-x    5 root    uucp          1024 Oct  2 17:14 /var/lock
```

尽管 Red Hat 6.2 已经过时，而且发行套件上的默认值也可能不一样，此处的设置亦不失为“进行必要的修改以获得串行端口适当使用权”的绝佳范例。就此处所举的例子来说，要以一般用户的身份在串行端口上使用终端仿真程序，必须是 tty 和 uucp 使用群组的成员之一，而且 `/dev/ttyS0` 的存取权限必须被改变为，让拥有群组（即使用群组）具备读和写的使用权。在某些发行套件中，虽然 `/dev/ttyS0` 的存取权限会有正确的设定，但是 `/var/lock` 的拥有群组却是 root。若是这样的话，可能会想要改变群组设定值，除非你想让一般用户成为 root 群组的成员之一，否则不建议这么做。

回到 Red Hat 6.2，让我们使用 `chmod` 来改变 `/dev/ttyS0` 的存取权限：

```
$ su
Password:
# chmod 660 /dev/ttyS0
# ls -al /dev/ttyS0
crw-rw----    1 root    tty          4,  64 May  5 1998 /dev/ttyS0
```

接着，使用 `vigr`（注 14）编辑 `/etc/group` 文件，将用户名称加入 uucp 和 tty 这两列设定中：

```
...
tty:x:5:karim
...
uucp:x:14:uucp, karim
...
```

最后，先从 root 用户身份注销，再从自己的账号注销，并且重新登录自己的账号：

```
# exit
$ id
uid=501(karim) gid=501(karim) groups=501(karim)
$ exit

Teotihuacan login: karim
Password:
$ id
uid=501(karim) gid=501(karim) groups=501(karim),5(tty),14(uucp)
```

注 14： 这是为了编辑 `/etc/group` 文件而定制的命令。它可以正确地锁定文件，以确保在任何时候只有一个用户可以存取该文件。详见其 manpage。

正如所见，首先需要注销，然后再重新登录，好让改变生效。在 GUI 中开启新的终端窗口可能也会有类似的效果。然而，尽管这么做有用，但只有新的终端窗口具有正确的群组设定，任何在变更之前开启的其他窗口，它们的群组设定仍是不正确的。因此，最好还是先离开 GUI，等完全注销后再重新登录。

想要进一步了解串行端口接口的设置，可参考 LDP 的“Serial HOWTO”以及《Linux Network Administrator's Guide》(O'Reilly) 的第四章。

minicom

minicom 是 Linux 上最常用的终端仿真程序。大多数与嵌入式系统有关的（不论是在线或印刷出来的）文件，都会假定你使用的是 minicom。然而，正如前面所说，已经知道 minicom 与至少一种引导加载程序之间有文件传输的问题。minicom 可说是 *Telnet* 这个 DOS 程序的 GPL 仿照版，它具备 ANSI 和 VT102 终端的能力。minicom 计划的网站目前位于 <http://alioth.debian.org/projects/minicom/>。minicom 可能已经被发行套件安装在主机上了。如果使用的是以 Red Hat 为基础的发行套件。可以利用 `rpm -q minicom` 这条命令看看是否已经安装了。

欲启动 *minicom* 可使用如下的命令：

```
$ minicom
```

minicom 随后会进入全屏幕模式，并在屏幕的顶端有如下的显示：

```
Welcome to minicom 1.83.0

OPTIONS: History Buffer, F-key Macros, Search History Buffer, 118n
Compiled on Mar  7 2000, 06:12:31.

Press CTRL-A Z for help on special keys
```

欲键入命令给 minicom，可先按 Ctrl-A 再按代表命令的字母。正如 minicom 的欢迎信息所提到的，使用 Ctrl-A Z 可以从 minicom 获得辅助信息。使用细节请参考该套件的 *manpage*。

UUCP cu

UUCP (Unix to Unix CoPy) 是连接 Unix 系统最常见的方法之一。尽管现在 UUCP 已经很少用到了，不过 UUCP 套件中的 *cu* 命令却可用来连通其他系统。与其他系统通信可以有許多连接形式。在此例中，我们最感兴趣的就是通过串行线路建立起与目标板之间的终端联机。

为此，我们必须在 UUCP 使用的配置文件中加入适当的设定条目。尤其是，这表示要在 */etc/uucp/port* 文件中加入端口的设定条目，以及在 */etc/uucp/sys* 文件中加入远程系统的定义。正如 UUCP 的 info page 所说“在计算机上，端口是一个特殊的硬件连接”，而系统定义则是用来描述所要连接的系统以及要用哪个端口来连接它。

尽管你只要同意 GPL 的许可条款就可以到 GNU FTP 网站下载 UUCP 套件，不过它通常已经安装在系统上了。在一个以 Red Hat 为基础的系统上，可以利用 *rpm -q uucp* 来看看它是否已经安装了。

下面是 */etc/uucp/port* 范例文件的内容：

```
# /etc/uucp/port - UUCP ports
# /dev/ttyS0
port      ttyS0      # Port name
type      direct     # Direct connection to other system
device    /dev/ttyS0 # Port device node
hardflow  false      # No hardware flow control
speed     115200     # Line speed
```

根据这些条目的设定我们知道：端口的名称为 *ttyS0*，它使用 115200 bps 的速率直接联机，没有硬件流量控制，它通过 */dev/ttyS0* 连接远程系统。端口的名称在此例为 *ttyS0*，此名称仅用来让其他的 UUCP 工具和配置文件辨识该端口的定义而已。然而，如果在使用 UUCP 之前会先用传统的调制解调器进行联机，那么设定项中就会加入类似调制解调器的定义。然而，不必使用 *carrier* 字段来指定是否应该收到载波。将连接类型设定成 *direct* 就可以让 *carrier* 被预设为 *false*。

下面是 */etc/uucp/sys* 范例文件的内容：

```
# /etc/uucp/sys - name UUCP neighbors
# system: target
system    target     # Remote system name
port      ttyS0      # Port name
time      any        # Access is possible at any time
```

根据这些条目的设定我们知道：远程系统称为 *target*，在任何时候都可以使用 *ttyS0* 这个端口与它联系。

现在我们可以使用 *cu* 来连接目标板：

```
$ cu target
Connected.
```

一旦 *cu* 进入联机状态，可以使用 *~* 字符来发布命令，其后要跟着另一个代表实际命令的符号。使用 *~?* 可列出完整的命令清单。

欲进一步了解如何为系统设定及定制 UUCP，请参考《Linux Network Administrator's Guide》(O'Reilly) 第十六章、LDP 的“UUCP HOWTO”以及 UUCP 的 info page。

C-Kermit

C-Kermit 是哥伦比亚大学的 Kermit 计划 (<http://www.columbia.edu/kermit/>) 中维护的套件之一。C-Kermit 广泛地为各种平台的网络操作提供了一致的接口。尽管它的特色是具有许多功能，不过终端仿真能力是我们对此套件最感兴趣的部分。

虽然你可以基于个人和内部使用自由下载 C-Kermit，但是 C-Kermit 并非开放源码软件，而且它的许可条款不同于包含它的商业发行套件（注 15）。欲获得 C-Kermit，可以从 <http://www.columbia.edu/kermit/ckermite.html> 下载。你可以根据该套件附带的 *ckuins.txt* 文件中的指示，编译并安装 C-Kermit。与我们在本书探讨的大多数其他工具相比，C-Kermit 应该安装在整个系统的工作空间之上，而不是安装在项目自己的工作空间。一旦安装好之后，可以使用 *kermit* 命令来启动 C-Kermit。

就可用性来看，*kermit* 与 *minicom* 和 *cu* 相比，站在相当有利的位置。尽管 *kermit* 缺乏 *minicom* 提供的用户菜单，但是当你与终端仿真程序交互时，*kermit* 的交互式命令语言却非常直接有效。例如，当你对目标板的引导加载程序发起文件传输作业时，引导加载程序便会开始等待文件的到来。接着你可以在主机上使用 Ctrl-AC 切换到 *kermit* 的交互式命令行，并且使用 *send* 命令送出实际的文件。此外，就像 Linux 中大多数的 shell，*kermit* 的交互式命令行也提供使用 Tab 键的文件名补全功能。同时，*kermit* 的交互式命令行还允许你使用命令的简写（可惟一代表该命令的最短字符串）来发布命令。例如，*set receive* 命令可以被简写成 *set rec*。

要使用 *kermit* 命令，必须将 *.kermrc* 配置文件放到主目录。*kermit* 启动时会加载此文件。下面是 *.kermrc* 范例文件的内容：

```
; Line properties
set modem type          none ; Direct connection
set line                /dev/ttyS0 ; Device file
set speed               115200 ; Line speed
set carrier-watch       off ; No carrier expected
set handshake           none ; No handshaking
set flow-control        none ; No flow control

; Communication properties
robust                  ; Most robust transfer settings macro
set receive packet-length 1000 ; Max pack len remote system should use
```

注 15： 尽管后来为了避免商业发行套件，例如 Red Hat，附带此套件有困难，变更过授权条款，不过并非所有的主流发行套件都会附带 C-Kermit。

```
set send packet-length      1000 ; Max pack len local system should use
set window                  10 ; Nbr of packets to send until ack

; File transfer properties
set file type                binary ; All files transferred are binary
set file names               literal ; Don't modify filenames during xfers
```

欲进一步了解上面每项设定值的意义，可试试 *kermit* 交互式命令行提供的 *help* 命令。例如，欲进一步了解 *robust* 宏，可以使用 *help robust*。此例中，必须在 *set receive* 之前使用 *robust*，因为 *robust* 是用来将远程系统使用的最大包长度设成 90 个字节，然而我们想将它设成 1000 个字节。

配置文件一旦设好之后，接着就可以启动 *kermit*：

```
$ kermit -c
Connecting to /dev/ttyS0, speed 115200
  Escape character: Ctrl-\ (ASCII 28, FS): enabled
Type the escape character followed by C to get back,
or followed by ? to see other options.
-----
```

如果正在寻找更多有关如何使用 C-Kermit 的信息，因为你想要更广泛地使用 C-Kermit，可以考虑购买由 Frank Da Cruz 与 Christine Gianone 合著的《Using C-Kermit》(Digital Press)。购买此书除了可以让你了解如何使用 C-Kermit，还可以让该计划受到资助。尽管此书只涵盖 6.0 版，不过你可以到该计划的网站上免费获得 7.0 和 8.0 版的补充资料。



内核方面的考虑

内核是所有 Linux 系统的中心软件组件。整个系统的能力完全受内核本身能力的限制。例如，倘若你使用的内核无法支持目标板上某个硬件组件，当在目标板上运行此内核时，该硬件组件将会变得毫无用处。

事实上，已经有许多书籍和在线文档在探讨内核的内部、程序设计、设置及其在用户系统上的使用，而且都很详细。因此本章不会涵盖这些问题。如果对这些问题有兴趣，不妨阅读《Running Linux》、《Linux Device Drivers》和《Understanding the Linux Kernel》等由 O'Reilly 出版的书。这些书分别涵盖了内核的设置与使用、内核的程序设计及内核的内部。你或许还会想看一下 LDP 的《Linux Kernel HOWTO》。

本章将专门讨论为了在嵌入式系统上使用 Linux 怎样准备内核。尤其是，我们将会探讨内核的选择、配置、编译及安装。每完成一个步骤我们就会越接近“让目标板系统获得可用内核和相关模块”这个目标。我们最后还会讨论嵌入式系统特有的内核操作的各个方面。

选择内核

尽管内核的主要供应网站只有 <http://www.kernel.org/>，但是从这个网站取得的内核版本不一定可以用在 Linux 支持的每个架构上。事实上，当你以这些版本为嵌入式 Linux 系统中最常见的架构建立内核时，有些架构甚至会失败，能正常执行的就更少了。主要是因为这些架构的 Linux 开发与主要的内核版本不同步。

要让目标板取得可用的内核，必须找到专门负责开发相应处理器架构的团队所提供的内核版本。因为每种架构都由不同的团队维护，所以你必须根据架构来选择供应内核的网站。表 5-1 列出了每种处理器架构最适合的内核供应网站以及下载方式。

表 5-1: 每种处理器架构最适合的内核供应网站

处理器架构	最适合的内核供应网站	下载方式
x86	http://www.kernel.org/	ftp, http, sync
ARM	http://www.arm.linux.org.uk/developer/	ftp, rsync
PowerPC	http://penguinppc.org/	ftp, http, rsync, bitkeeper
MIPS	http://www.linux-mips.org/	cvs
SuperH	http://linuxsh.sourceforge.net/	cvs
M68k	http://www.linux-m68k.org/	ftp, http

正如所见, 这些大部分是我在第三章提到每种架构时介绍的网站。有人说, 要取得这些架构的内核, 不一定要到这几个地方, 别处也可以。有两种情况。首先, 这些网站有的会有镜像网站, 镜像网站提供的是相同的内容。其次, 有许多人、公司和组织提供自己开发的内核版本。如果使用的是后者, 千万要小心, 这些内核可能不会得到社群的支持(注1), 当有问题发生时, 就算有也只限于供应者提供的支持。

一旦找到了最适合的下载网站, 将需要从该网站选出适用的内核版本。这是个困难的抉择, 因为有些版本的某些功能有问题, 但是在较旧的版本中这些功能又完全正常。找到这类信息的最佳办法就是, 不断地跟维护架构内核的社群接触。这并不表示你要接触或写信给任何人, 而是指订阅适当的邮件论坛并且不断地检查论坛上和该项移植的主要网站上的重要公告。

这些网站有的, 例如ARM的网站, 未必会发布完整的内核, 而是发布正式内核的补丁。此时, 如果要为架构取得可用内核, 必须先到主要供应网站下载内核, 然后再用移植内核的网站提供的补丁对内核进行修补。

为了实现以ARM为基础的用户接口, 我们会从<http://www.kernel.org/>下载一般的2.4.18内核, 并从正式的ARM Linux网站<http://www.arm.linux.org.uk/>下载2.4.18-rmk5补丁。用rmk5修补过一般的2.4.18之后, 我们便得到了2.4.18-rmk5内核, 其中包含了以ARM为基础的系统所需要的功能。

注1: 社群之所以不支持, 未必是因为没有源码可用(因为Linux的发行受到GPL授权条款的保护, 所以不应该发生此事)、最可能是因为供应者对内核机能所做的修改, 只有供应者自己知道。也可能是供应者对内核所做的修改, 被社群认为不够成熟或甚至不需要加入主内核树中

内核版本编号的变化

其他内核供应网站发行的版本，通常会将内核编号系统加以变化，以区分其产品。ARM 内核源码树的维护者 Russell King，会为自己发行的内核加上 -rmk 的扩展名。以 Russell 的成果为基础的其他开发者，则会为他们所发行的内核加上自己的扩展名。另一个 ARM Linux 的开发者 Nicolas Pitre 会为自己的内核加上 -np 的扩展名，而 handhelds.org 的 Familiar 发行套件的开发者们，则会为他们的内核加上 -hh 的扩展名。因此，我在第一章所提到的 2.4.20-rmk3-hh24 这个版本，事实上是：Russell 以 Marcelo Tosatti 的 2.4.20 版本为基础修改的内核，在第 3 次发行时，handhelds.org 的开发者们以这个版本（-rmk3）为基础对内核进行了修改，并且发行了 24 次（-hh24）。

（尽管 Linus Torvalds 通常是 Linux 版本的维护者，但是 Linus 将 2.4.x 系列的维护责任交给了 Marcelo，好让自己全力投入 2.5.x 系列的开发工作。）

通常情况下，选择已知可用的版本中越新的版本越好。因此，如果已知在目标板上可以使用 2.4.17 和 2.4.18，则 2.4.18 应该是最优选的版本。然而，在某些情况下并非如此。例如，密切注意内核开发的人大都知道，应该避免使用版本 2.4.10 到 2.4.15（含）的内核，因为它们是在许多修改正要被整合进内核的过程中产生的版本，因此偶尔会有不稳定的现象。再次提醒，要取得这类信息，必须持续地接触适当的邮件论坛及网站。

如果发现订阅移植版本的邮件论坛或主内核版本的邮件论坛太浪费时间，应该找时间至少一周探访一次移植版本的网站及阅读《Kernel Traffic》(<http://www.kerneltraffic.org/>) 周报。《Kernel Traffic》摘录了上周主内核版本邮件论坛中最重要的讨论内容。

一旦为目标板找到了适当的内核版本之后，可以像第四章“内核头文件的设置”中那样，先将它下载到 $\{PRJROOT\}/kernel$ 目录并从中取出源码，如果有需要再为它更名。为内核目录更名，可以避免日后从所下载的另一个内核版本取出源码时，误将既有的内核目录覆盖掉。

不论你要选用哪个版本，务必为目标板多试几个不同的内核版本。除了可以参考网络上的建议和缺陷报告，多评估几个不同的版本将有助于你洞悉硬件与内核间的相互作用。

你或许还想尝试其他开发者提供的各种补丁。额外的内核功能被整合进主流内核之前通常是以独立的补丁存在的。例如，Robert Love 对内核的抢占功能所做的修补，在被 Linus 整合进 2.5 开发版系列之前，被制作成独立的补丁。我们将会在第 11 章探讨如何对内核进行修补。如果对修补的事情不熟悉的话，可参考《Running Linux》(O'Reilly) 这本书。

内核配置

在你为目标板建立内核的过程中，配置属于最初的阶段。内核配置的方法很多，而且配置时有许多选项可以选择。

不管你使用哪种方法来设定配置或选择哪些配置选项，在你设好配置之后内核都将会产生 `.config` 文件以及建立过程其余步骤将会用到的一些符号链接和头文件。

以下的讨论将会局限于在嵌入式系统中配置内核有什么不同。如果对内核配置不熟悉的话，可查阅我前面所提到的各种参考资源。

配置选项

配置设定期间，想纳入内核的功能可以通过选项来选择。因目标板而定，所看到的选项可能不一样。然而，有些选项不管你选用哪种嵌入式架构都会存在。以下列出所有嵌入式 Linux 架构都看得到的主菜单选项：

- Code maturity level options
- Loadable module support
- General setup
- Memory technology devices
- Block devices
- Networking options
- ATA/IDE/MFM/RLL support
- SCSI support
- Network device support
- Input core support
- Character devices
- Filesystems
- Console drivers
- Sound
- Kernel hacking

我不准备深入探讨每个选项，因为内核配置菜单提供有辅助说明的功能，可以在设定配置的同时参考它。然而请注意，我们在第三章已经讨论过其中不少选项。

菜单中最重要的就是你用来为目标板选择处理器架构的选项。然而此选项的名称因架构而异。表 5-2 根据各种处理器架构列出了系统和处理器选项的名称以及正确的内核架构名称。在我们执行 *make* 命令的时候，需要将 ARCH 的值设成内核的 Makefile 中定义的正确架构名称。

表 5-2: 根据处理器架构列出系统和处理器选项及内核架构的名称

处理器架构	系统和处理器选项	内核架构名称
x86	Processor type and features	i386
ARM	System type	arm
PPC	Platform support	ppc
MIPS	Machine selection/CPU selection	mips 或 mips64 ^a
SH	Processor type and features	sh
M68k	Platform-dependent support	m68k

a. 取决于 CPU。

有些选项只会在特定的架构中出现。表 5-3 列出了这些选项在内核配置菜单中的名称，并指出它们会出现在哪些架构上。

表 5-3: 每种架构的硬件支持选项

选项	x86	ARM	PPC	MIPS	SH	M68k
Parallel port support	×	×		×		
IEEE 1394 support	×	×	×		×	
IrDA support	×	×	×	×		
USB support	×	×	×	×		
Bluetooth support	×	×	×			

有些架构具有专用的配置选项。下面列出的是 ARM 架构特有的选项：

- Acorn-specific block devices
- Synchronous serial interfaces
- Multimedia capabilities port drivers

下面则是 PPC 特有的选项：

- MPC8xx CPM options
- MPC8260 communication options

事实上，即使某个选项出现在架构的配置菜单中，并不代表目标板支持此功能。更确切地说，配置选项可能会允许你启用目标板未曾测试过的许多功能。例如，ARM 系统上没有 VGA 控制台。然而，内核的配置菜单却允许你启用 VGA 控制台的支持。因此，如果启用了这个选项，建立内核将会失败，或者造成所选用的功能甚至是整个内核毫无作用。要避免发生这类问题，确定目标板支持你启用的选项。大部分时候，就像 VGA 控制台那样，这是一般常识。当对某些选项不是很清楚的时候，可以访问相关计划的网站，例如第三章提到的网站将有助于你判断目标板是否支持此功能。

有些时候，即使某个选项并未在架构的配置菜单中显示，并不代表此功能无法在目标板上使用。表 5-3 列出了许多此类功能，例如 Bluetooth，它们大多数与架构无关，在任何架构上运行都应该没有问题。它们之所以未列在特定架构的配置菜单中，不是因为它们没有在那里测试过，就是因为那些移植成果或功能的维护者并未要求将此功能加入内核的因架构而异的 *config.in* 文件中（注 2）。再次提醒，如果想找出目标板可能支持哪些未列出的功能，第三章提到的参考资源是很好的起点。

设定配置的方法

内核支持四种设定配置的方法：

make config

通过命令接口，依次要求你设定每个选项。如果 *.config* 配置文件存在，它会根据该文件来设定（你设定的选项的）默认值。

make oldconfig

通过命令接口，但是会自动馈入既有的 *.config* 配置文件，并且只有在遇到先前没有设定过的选项时，才会要求你手动设定。然而，*make config* 却会要求你手动设定所有的选项，即使你之前曾设定过。

make menuconfig

显示以 *curses* 为基础的、终端式的配置菜单。如果 *.config* 文件存在，它会根据该文件来设定默认值，如同 *make config* 一样。

注 2： *config.in* 文件用来控制配置菜单应该显示哪些选项。

make xconfig

显示以 Tk 为基础的 X Window 配置菜单。如果 *.config* 文件存在，它会根据该文件来设定默认值，如同 *make config* 和 *make menuconfig* 一样。

以上四种方法都可以用来设定内核配置。它们都会在内核源码的根目录中产生 *.config* 文件。（此文件包含了你对选项所做的设定的全部细节。）

只有少数开发者会使用 *make config* 来设定内核配置。一般开发者通常会使用 *make menuconfig*。你也可以使用 *make xconfig*。然而不要忘了，在某些架构中，例如 PowerPC，使用 *make xconfig* 时，菜单可能有问题。

要检查内核配置菜单，只须在命令行上键入适当的命令以及恰当的参数。对这个以 ARM 为基础的用户接口模块，我们可以使用如下的命令行：

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- menuconfig
```

然后我们必须选择适合目标板使用的配置选项。许多功能和驱动程序可以以模块的形式存在，我们可以选择是否将它们建立在内核中，或是将它们建立成模块的形式。一旦设定好内核配置之后，我们可以使用 Esc 键或是选择 Exit 条目离开配置菜单。然后配置工具程序会问我们是否保存配置。我们可以选择 Yes 条目来保存内核的配置以及建立 *.config* 文件。这除了会建立 *.config* 文件，也会建立一些的头文件和符号连接。如果选择 No，则不会保存我们设定的配置，也不会改变既有的配置。

除了主要的配置选项，有些架构（例如 PPC 和 ARM）的配置设定，也可以使用为各种目标板实现时采用的架构而定制的配置。因此，它们提供给内核的默认值将会用来产生 *.config* 文件。举例来说，我会以如下的命令来为 TQM860L PowerPC 目标板设定内核配置：

```
$ make ARCH=ppc CROSS_COMPILE=powerpc-linux- TQM860L_config  
$ make ARCH=ppc CROSS_COMPILE=powerpc-linux- oldconfig
```

管理多种配置

我们通常都需要针对相同的内核源码测试不同配置。然而，如果改变内核的配置则会破坏先前的配置，这是因为所有配置文件都会被内核的配置工具程序（命令脚本）给覆盖掉。要将配置保存起来供日后使用，我们需要将内核的配置工具程序建立的 *.config* 文件保存起来。这些文件日后可用于恢复先前设定的内核配置。

备份及恢复配置的最简单方法就是使用内核自己的配置设定程序。*menuconfig* 和

xconfig这两个Makefile目标显示的菜单，可以让你保存和恢复配置。不论保存和恢复，都需要提供适当的文件名。

你还可以手动保存`.config`文件。方法是将内核配置命令脚本产生的配置文件（即`.config`文件）复制到其他地方供日后使用。要使用已保存的配置，必须将先前手动保存的`.config`文件复制回内核源码树的根目录，然后用`make`命令及`oldconfig`这个Makefile目标使用刚才复制的`.config`文件来设定内核的配置。如同`menuconfig`这个Makefile目标，`oldconfig`会产生若干头文件及符号链接。

不管你是手动或者是使用其他工具程序提供的菜单来复制文件，配置应该保存在凭直觉就可以找到的地方，并且使用有意义的命名机制来存放配置。以我们的项目工作空间为例，建议将所有的配置保存到`${PRJROOT}/kernel`目录，这样既可以独立于实际的内核源码之外，又可以表示它们是属于内核的数据。要区分每个配置文件，可以在其文件名前置相应内核版本的编号以及简单的描述或日期，或二者。最后，让`.config`变成扩展（副）文件名，这样一看便知道这是个内核配置文件。

以我们所使用的2.4.18版内核为例，我为它设定了一个不提供串行端口支持的配置。因此我将相应的配置文件称为`2.4.18-no-serial.config`。我还保存了一个到目前为止被认为是“最佳”的配置，并将它称为`2.4.18.config`。你可以随意采用任何你认为最直观的命名习惯，不过你可能要避免通称形式的名字，例如`2.4.18-test1.config`。

使用 EXTRAVERSION 变量

如果使用的是同一个内核版本的多个变体，会发现在你想要辨别每个实体时EXTRAVERSION变量非常有用。EXTRAVERSION变量的值会被附加在内核的版本编号之后，成为内核建立完成后的最终版本编号。例如，为2.4.18版的内核加入rmk5的补丁后，EXTRAVERSION变量的值会设成`-rmk5`，该内核建立完成后的最终版本编号为`2.4.18-rmk5`。

这个最终的版本编号还会当做一个目录的名称，你为内核建立的模块将会存放到此目录中。因此，当你为同一个内核版本建立两次模块时，如果EXTRAVERSION变量前后两次的值不同，则前后两次产生的模块将会存放不同的目录；如果前后两次都没有为EXTRAVERSION变量设值，则前后两次产生的模块将会存放相同的目录。

你还可以使用这个变量来区分基于同一个内核版本的各个变体。方法是编辑内核源码主目录中的Makefile文件，并将EXTRAVERSION变量设成你想要的值。EXTRAVERSION变量的值还有一个用法，就是你可以根据此值来为内容已改变的内核源码主目录改名。例如，倘若2.4.18版内核的EXTRAVERSION变量被设成`-motor-diff`，那么内核源码的

主目录应该被改名为 *2.4.18-motor-diff*。而你备份的 *.config* 文件应该使用 *EXTRAVERSION* 变量的值来命名。如果此内核的配置被设成不提供串行端口的支持, 则其配置文件应该被命名为 *2.4.18-motor-diff-no-serial.config*。

编译内核

内核的编译包括以下几个步骤: 建立内核源码的依存关系, 建立内核映像, 以及建立内核模块。以上每个步骤使用的 *make* 命令都不同, 以下我们会分节说明这几个步骤。当然, 也可以使用命令行来执行这几个步骤。

建立依存关系

内核源码树中大多数文件都会与一些头文件有依存关系。要想顺利建立内核, 内核源码树里各个 *Makefile* 必须知道这些依存关系。依存关系建立期间会在内核源码树中每个子目录里产生一个隐藏的 *.depend* 文件。此文件内含子目录里各文件所依存的头文件清单。如同其他靠 *make* 建立的软件, 自从上一次完成建立以来, 如果要重新建立内核, 只有在与头文件有依存关系的文件被改动后才需要经过重新编译的程序。

你可以从内核源码树的根目录, 以如下的命令来建立内核源码的依存关系:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- clean dep
```

这么做是因为前面在设定内核配置的时候, 我们曾设定 *ARCH* 和 *CROSS_COMPILE* 变量。正如第四章所说, 只有在实际编译源码的时候才需要设定 *CROSS_COMPILE* 变量。相对而言, 我们至少需要为自己发布的每道 *make* 命令设定 *ARCH* 变量, 因为我们正在进行内核的交叉编译。就算发布的是 *make clean* 或 *make distclean* 命令, 我们还是需要设定此变量。否则内核源码树里各个 *Makefile* 会假定, 目前对内核源码所进行的操作都跟主机的架构有关。

ARCH 变量用来指明要为哪种架构建立内核。内核源码树里各个 *Makefile* 会根据此变量来选择它准备使用的与架构相关的目录。当你为目标板编译内核的时候必须将此变量设成目标板的架构。

内核源码树里各个 *Makefile* 会根据 *CROSS_COMPILE* 来产生一些工具程序的名称, 这些工具程序将会用在建立内核的过程中。例如 C 编译器的名称就是 *CROSS_COMPILE* 变量的值与 “gcc” 串接而成的。以 ARM 目标板为例, C 编译器的最后名称为 *arm-linux-gcc*, 这就是我们根据第四章的指示建立的 C 编译器的实际名称。这也说明了, 为什么之前的命令行中结尾的连字号 (-) 如此重要的缘故。少了这个连字号, *Makefile* 所使用的编译器将会是 *arm-linuxgcc*, 这个编译器根本就不存在。

建立依存关系所需的时间相对较短。在我的PowerBook上，这个工作只需要2分钟的时间。这个阶段通常不会看到任何错误信息。如果看到了错误信息，表示内核可能遇到重大问题（译注1）。

建立内核

建立依存关系后，接着编译内核映像：

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- zImage
```

zImage这个建立目标用来指示Makefile建立经gzip算法压缩过的内核映像（注3）。不过还有其他方法可用来建立内核映像。例如vmlinux这个建立目标可用来指示Makefile只建立未经压缩的映像。请注意，当你要求建立经过压缩的映像时，也会产生这个未经压缩的映像。

在x86上，还可以使用bzImage这个建立目标。bzImage是big zImage的简写，它与bzip2压缩工具程序无关。事实上，bzImage和zImage这两个建立目标都是使用gzip算法。差别在于zImage产生的经压缩的内核映像无法超过512 KB，而bzImage则不受这个限制。如果想要进一步了解zImage与bzImage的差异，可以检查内核源码树包含的*Documentation/i386/boot.txt*文档。

内核配置设定期间，如果选择了架构不支持的选项或一些有问题的内核选项，建立工作将会在此阶段失败。如果一切顺利，整个建立过程所花的时间应该会比建立依存关系多几分钟。以我的硬件配置来说，整个过程需要5分钟的时间。

验证交叉开发工具链

请注意，建立内核让我们能够实际测试前一章建立的交叉开发工具。如果前面建立的工具能够成功编译出可用的内核，这代表所有其他软件的建立应该都没问题。当然，将需要下载内核源码、为目标板建立内核，以便验证其可用性，事实上，能够正确建立，已经具有正面意义了。

译注1： 例如找不到编译器。

注3： 尽管对我们在第三章深入探讨的所有架构来说，zImage是个有效的Makefile target（建立目标），但是对其他的Linux架构来说则不然。

建立模块

内核映像正确建立后，接着建立内核模块：

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- modules
```

这个阶段的时间长短与“你选择不要链接成主内核映像的一部分而要建立成模块的内核选项的数量”有很大的关系，这个阶段的时间很少会长过建立内核映像的时间。如同内核映像一样，倘若配置不符合目标板的需要或是内核选项有问题，这个阶段也可能会失败。

内核映像和内核模块都正确建立之后，表示我们已经准备好可以为目标板进行安装工作了。有一件事很重要，在进行之前要特别注意，如果需要清理内核的源码，让它恢复到配置设定、依存关系建立或编译之前的初始状态，可以使用如下的命令：

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- distclean
```

但务必在执行此命令之前将内核的配置文件备份起来，因为 *make distclean* 会清除前面这几个阶段产生的文件，包括 *.config* 文件、所有目标文件以及内核映像。

安装内核

最后，我们产生的内核映像以及它的模块必须复制到目标板上去。我将会在第六和第九章说明，如何复制内核映像以及它的模块。这里要讨论的是，如何管理多个内核映像以及相应模块的安装。目标板引导配置的安排以及它的根文件系统跟我们底下要讨论的技术有关。

管理多个内核映像

除了使用分开的目录来存放不同的内核版本，还会发现能够“使用多个内核映像来测试目标板”很有用。因为这些映像可能是由相同的源码建立的，所以我们需要将它们从内核源码树复制到不同的目录中。以我们的设置来说，这些映像会被存放到 *\$(PRJROOT)/images* 目录。

对每个内核配置来说，我们将需要复制四个档案：压缩的内核映像、未压缩的内核映像、内核符号映射文件以及配置文件。最后三个文件可以在内核源码树的根目录中找到，它们的文件名分别是 *vmlinux*、*System.map* 和 *.config*。而未压缩的内核映像则可在 *arch/YOUR_ARCH/boot* 目录中找到（而 *YOUR_ARCH* 就是目标板架构的名称），其文件名可能是 *zImage* 或 *bzImage*，这取决于你前面使用的 Makefile 建立目标。对使用 ARM 的目标板来说，未压缩的内核映就是 *arch/arm/boot/zImage*。

有些架构（例如 PPC）具有多个引导目录。在这种情况下，所要使用的内核映像，未必会放在 *arch/YOUR_ARCH/boot/zImage*。以之前提到的 TQM 板为例，压缩的内核映像应该会放在 *arch/ppc/images/vmlinux.gz*。根据目标板来检查 *arch/YOUR_ARCH/Makefile* 文件，可以看到各种引导映像的 Makefile 建立目标的完整细节。以我们的 PPC 范例来说，其产生的引导映像类型取决于编译内核时所针对的那个处理器型号。

为了区分这四个文件，我们使用与内核版本编号类似的命名机制。例如，倘若内核映像是由 2.4.18-rmk5 版的源码产生的，我们会以如下的方式复制这四个文件：

```
$ cp arch/arm/boot/zImage ${PRJROOT}/images/zImage-2.4.18-rmk5
$ cp vmlinux ${PRJROOT}/images/vmlinux-2.4.18-rmk5
$ cp System.map ${PRJROOT}/images/System.map-2.4.18-rmk5
$ cp .config ${PRJROOT}/images/2.4.18-rmk5.config
```

你还可以在文件名中纳入配置名称。因此在内核不提供串行支持的情况下，我们可以将这四个文件称为 *zImage-2.4.18-rmk5-no-serial*、*vmlinux-2.4.18-rmk5-no-serial*、*System.map-2.4.18-rmk5-no-serial* 和 *2.4.18-rmk5-no-serial.config*。

安装内核模块

内核源码的 Makefile 文件包含的 `modules_install` 建立目标可用来安装内核模块。缺省情况下，内核模块会安装到 */lib/modules* 目录。然而，因为我们使用的是交叉开发环境，所以我们必须指示 Makefile 将模块安装到另一个目录。

因为内核模块由相应的内核映像使用，所以我们会把模块安装到名称与内核映像类似的目录。以我们使用的 2.4.18-rmk5 内核为例，我们会把模块安装到 *\${PRJROOT}/images/modules-2.4.18-rmk5* 目录。此目录的内容稍后将会被复制到目标板的根文件系统，供目标板上相应的内核使用。我们会以如下的方式将模块安装到该目录：

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- \
> INSTALL_MOD_PATH=${PRJROOT}/images/modules-2.4.18-rmk5 \
> modules_install
```

/lib/modules 路径会前置 `INSTALL_MOD_PATH` 变量的值，因此模块会被安装到 *\${PRJROOT}/images/modules-2.4.18-rmk5/lib/modules* 目录。

一旦复制好模块之后，内核就会根据模块工具程序执行时的要求，试着建立模块依存关系。因为 *depmod* 这个用来建立模块依存关系的工具程序，并非用来处理交叉编译的模块，所以会执行失败。

要为模块建立模块依存关系必须用到BusyBox提供的模块依存关系建立程序。BusyBox在第六章将会有深入的讨论。现在我们可以从<http://www.busybox.net/>将BusyBox包的副本下载到`${PRJROOT}/sysapps`目录，并在该处取出源码（注4）。接着我们可以从存放BusyBox源码的目录将`scripts/depmod.pl`命令脚本复制到`${PREFIX}/bin`目录。

现在我们可以为目标板建立模块依存关系：

```
$ depmod.pl \  
> -k ./vmlinux -F ./System.map \  
> -b ${PRJROOT}/images/modules-2.4.18-rmk5/lib/modules > \  
> ${PRJROOT}/images/modules-2.4.18-rmk5/lib/modules/2.4.18-rmk5/modules.dep
```

其中，`-k`选项用来指定未压缩的内核映像、`-F`选项用来指定系统对映文件、`-b`选项用来指定内核需要建立依存关系的模块的基本目录。因为此工具的执行结果会送往标准输出，我们可以将它重定向到一个文件中，其文件名总是叫做 *modules.dep*。

实地测试

当内核安装在目标板上准备执行的时候，让我们检查内核的运行状态。因为嵌入式系统的算法以及底下的源码跟一般系统没两样，所以内核在嵌入式系统上的运行几乎跟在工作站和服务器的完全一样。因此，对内核有更深入探讨的其他书籍和在线资料，例如O'Reilly出版的《Linux Device Drivers》和《Understanding the Linux Kernel》并未特别强调它们也适用于嵌入式Linux系统。

当内核失败时

Linux内核是一个非常稳定且成熟的软件。然而这并不表示Linux或Linux使用的硬件永远不会出故障。《Linux Device Drivers》一书便探讨oops信息和系统挂起之类的问题。除了需要在系统设置期间留意这些问题，还应该考虑到内核失败的最常见形式：内核恐慌。

当发生严重的错误而且被内核捕捉到时，内核将会停止所有的进程并且送出内核恐慌信息。发生内核恐慌的理由很多。其中最常见的是，忘了为内核指定它的根文件系统的位置。在这种情况下，内核将会正常引导，但是会在尝试安装根文件系统的时候发生内核恐慌。要从内核恐慌错误中恢复，唯一的办法就是让系统重新引导。因此内核可以接受用来指定“内核应该在内核恐慌之后几秒重新引导”的引导参数。例如，倘若你想让内核在内核恐慌之后1秒重新引导就应该使用`panic=1`这个内核引导参数。

注4. 请下载BusyBox 0.60.5或之后的版本。

然而有时就算是重新引导可能还不够，这取决于我们的设置。例如我们的控制模块，重新引导甚至可能导致危险，因为它控制的化学或机械程序可能会失常。因此我们需要修改内核的 panic 函数，让它通知操作人员使用紧急程序手动控制系统。当然，系统对内核恐慌的实际反应，取决于系统的实际应用。

内核的 panic 函数（即 panic()）的程序代码就放在内核源码树里的 *kernel/panic.c* 文件中。我们首先注意到，缺省情况下，panic 函数的执行结果会输出到控制台（注 5）。因为系统可能连终端都没有，所以你可能会想要根据系统特殊的硬件需求来修改该函数。例如你可以将实际的错误信息字符串写入闪存中你特别为此用途保留的区段。在系统重新引导之后，就能够从该闪存区段取回文字信息，并根据此信息解决问题。

不论你是否对实际的文字信息感兴趣，都可以向内核注册自己的 panic 函数。该函数会在某个内核恐慌事件发生时，被内核的 panic 函数调用，可用来进行发出警报、通知紧急事件之类的事。

被内核自己的 panic 函数调用的其他 panic 函数，会被放在 panic_notifier_list 列表中。notifier_chain_register 函数可用来将一个条目（其他 panic 函数）加入此列表中。相反地，notifier_chain_unregister 函数可用来将一个条目从此列表中移除。

你自己的 panic 函数摆在哪里不是很重要，但是务必在系统初始化期间完成函数的注册工作。以我们的例子来说，*mypanic.c* 文件会被放在内核源码树里的 *kernel/* 目录中，我们会依需要修改该目录的 Makefile。下面是供我们的控制模块使用的 *mypanic.c* 文件：

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/notifier.h>

static int my_panic_event(struct notifier_block *,
                          unsigned long,
                          void *);

static struct notifier_block my_panic_block = {
    notifier_call:    my_panic_event,
    next:             NULL,
    priority:         INT_MAX
};

int __init register_my_panic(void)
{
    printk("Registering buzzer notifier \n");

    notifier_chain_register(&panic_notifier_list,
                           &my_panic_block);
}
```

注 5： 所有系统信息都会送往控制台这个主要的终端。

```
        return 0;
    }

    void ring_big_buzzer(void)
    {
        ...
    }

    static int my_panic_event(struct notifier_block *this,
                             unsigned long event,
                             void *ptr)
    {
        ring_big_buzzer();

        return NOTIFY_DONE;
    }

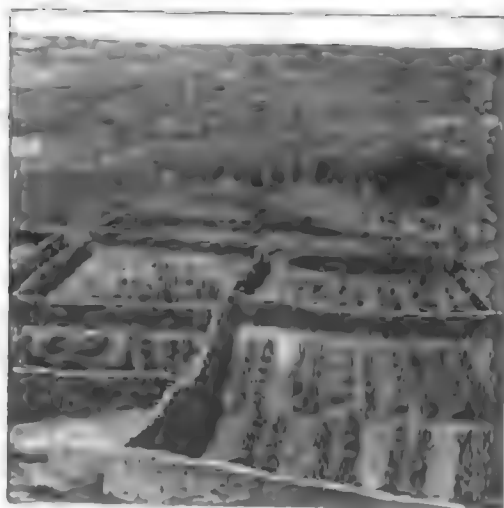
    module_init(register_my_panic);
```

`module_init(register_my_panic);`这个语句使得在内核初始化期间就可以调用 `register_my_panic` 函数，因此不必对内核的启动函数做任何修改。注册函数会在 `panic_notifier_list` 列表中加入 `my_panic_block` 条目。`notifier_block` 结构具有三个字段。第一个字段是所调用的函数，第二个字段会指向下一个 `notifier_block` 条目，第三个字段则是此条目的优先权。此例子中，我们想要让它具有最高的优先权，因此会使用 `INT_MAX` 这个值。

当发生内核恐慌时，`my_panic_event` 函数会被调用，成为所有 panic 函数的内核通告的一部分。接下来，`my_panic_event` 会调用 `ring_big_buzzer`，函数里的程序代码会启动响亮的警报器，让操作员注意到即将发生的问题。

第六章

根文件系统的内容



Linux 内核在系统启动期间进行的最后操作之一就是安装根文件系统。根文件系统一直都是所有 Unix 系统不可或缺的组件。根文件系统目前的结构有点独特，而且包含了一些多余之处，这是因为它与日俱进的成长深受 Unix 发展的影响，我并无意解释为何会形成目前的结构及惯例，我只打算说明如何遵照公认的标准来组织各种组件，由此产生可用的根文件系统。在这个过程中，我们将会用到前面建立的许多组件，例如内核模块和 C 链接库。

我们首先会探讨根文件系统的基本结构，然后我们会探讨如何安装系统链接库、内核模块、内核映像、设备节点、主系统应用程序以及定制应用程序以及把他们安装到何处。最后我们会讨论如何设定系统的初始化命令行。本章结束时你可以为目标板建立起一个全功能的根文件系统。接着我们会在下一章探讨如何将根文件系统实际地保存到存储设备上供目标板使用。

根文件系统的基本结构

根文件系统的顶层目录各有其特殊的用法和目的。其中有许多目录只对多用户系统有意义，而这些由不同用户使用的服务器及（或）工作站，则是由系统管理员负责维护。在大多数的嵌入式 Linux 系统中，通常不会有所谓的用户和管理者，所以不必严格看待用来建立根文件系统的规则，这并不表示你可以违反所有的规则，只是意味着破坏某些规则对系统的正常操作几乎没有影响。值得注意的是，即使是用于工作站和服务器的主流商业发行套件也不一定会遵守根文件系统的建立规则。

用来建立根文件系统的“正式”规则列在第一章提到的 Filesystem Hierarchy Standard (FHS) 中。这个文档的篇幅不超过 30 页，而且相当容易阅读，如果在寻求与根文件系统有关的答案或说明，FHS 或许是绝佳的起点。表 6-1 提供了根文件系统顶层目录的完整清单以及 FHS 指定的内容。

表 6-1: 根文件系统顶层目录

目录	内容
<i>bin</i>	必要的用户命令（二进制文件）
<i>boot</i>	引导加载程序使用的静态文件
<i>dev</i>	设备文件和其他特殊文件
<i>etc</i>	系统配置文件，包括启动文件
<i>home</i>	用户主目录，包括供服务账号所使用的主目录，例如 FTP
<i>lib</i>	必要的链接库，例如 C 链接库、内核模块
<i>mnt</i>	安装点，用于暂时安装文件系统
<i>opt</i>	附加的软件套件
<i>proc</i>	用来提供内核与进程信息的虚拟文件系统
<i>root</i>	root 用户的主目录
<i>sbin</i>	必要的系统管理员命令（二进制文件）
<i>tmp</i>	暂时性的文件
<i>usr</i>	在第二层包含对大多数用户都有用的大量应用程序和文件，包括 X 服务器
<i>var</i>	监控程序和工具程序所存放的可变数据

如果 Linux 是你每天必用的工作平台，应该已经熟悉以上提到的某些目录。不过，现在让我们来进一步检查嵌入式 Linux 系统如何使用这些典型的根文件系统内容。

首先，为多用户提供可扩展环境的所有目录（例如 */home*、*/mnt*、*/opt* 和 */root*）都应该省略。调整根文件系统的时候，我们甚至可以进一步移除 */tmp* 和 */var*，不过这么做可能会危害到某些软件的运行。因此不建议采用这种过于简化的做法。

注意： 此处的讨论重点并不在“大小”这个议题上，而是在“功能”上。事实上，省略某个目录只会对根文件系统的大小造成一些影响。理由是：例如，我说 */home* 可以省略，但就算嵌入式 Linux 系统中保留它，它也是空的，因为根据 FHS 的规定，它的内容应该只对工作站和服务器的设置有用。

根据引导加载程序和它的配置情况，可能不需要 */boot* 目录。这取决于引导加载程序是否会在内核被启动之前从根文件系统取回内核映像。读过第九章之后，将能够决定目标板是否应该使用 */boot* 目录以及如何使用它。当然，日后如果觉得有此需要，还可以重新设计根文件系统。

其余的目录，*/bin*、*/dev*、*/etc*、*/lib*、*/proc*、*/sbin* 和 */usr*，都是不可或缺的。

极端情况下你可以省略`/proc`，因为它只能用来安装与其同名的虚拟文件系统。然而这么做之后，如果需要实地分析目标板，将会很难了解目标板发生了什么事。如果为了缩减存储空间，可能会将内核设定成不支持`/proc`，但是无论如何还是建议尽可能启用此功能。

`/usr`和`/var`这两个顶层目录与根目录非常像，有自己的目录结构。在接下来的步骤中，当我们在摆放这两个目录的时候，将会简述它们的目录结构。

令人困惑的相似性

根文件系统最令人困惑的一点就是有些目录看起来具有类似的用途。尤其是，新手常会问，不同目录包含的二进制文件，以及不同目录包含的链接库，有何差异。

在根文件系统上，存放二进制文件的目录主要有四个：`/bin`、`/sbin`、`/usr/bin`和`/usr/sbin`。二进制文件要放在其中哪个目录，这与它在系统中所扮演的角色有很大的关系。如果这是用户和系统管理员必备的二进制文件，就会放在`/bin`，如果这是系统管理员必备、但是一般用户根本不会用到的二进制文件，就会放在`/sbin`。相对而言，如果不是用户必备的二进制文件，多半会放在`/usr/bin`；如果不是系统管理员必备的工具，多半会放在`/usr/sbin`。

至于链接库的摆放位置，也是同样的道理。系统引导以及执行最基本命令需要的链接库会摆在`/lib`。所有其他的链接库则会摆在`/usr/lib`。通常，套件安装时，会在`/usr/lib`中产生子目录，以便摆放它自己的链接库。以Perl 5.x为例，它会产生`/usr/lib/perl5`目录，里面摆放的都是与Perl有关的链接库和模块。

回过头来看看自己使用的Linux工作站，可以从它的根文件系统看到，发行套件设计者应用这些标准的实际范例。

为了建立根文件系统，让我们移往针对此目的所设置的目录：

```
$ cd $(PRJROOT)/rootfs
```

现在我们可以针对系统的需要建立根文件系统的顶层目录：

```
$ mkdir bin dev etc lib proc sbin tmp usr var
$ chmod 1777 tmp
```

请注意，我们并未建立`/boot`。如果日后需要再建立也不迟。同时请注意，我们变更了`/tmp`目录的使用权，让它开启sticky位。为`/tmp`目录的使用权开启此位，可确保`/tmp`目录下建立的文件，只有建立它的用户有权删除。尽管正如我之前所说，嵌入式系统多

半是单用户系统，不过有些嵌入式应用一定不能用root的特权来执行，因此需要遵照根文件系统权限位的一些基本规定。例如第十章探讨的 OpenSSH 套件便属于此类应用。

接着我们可以建立 */usr* 的目录结构：

```
$ mkdir usr/bin usr/lib usr/sbin
```

在一个全功能的根文件系统上，*/usr* 目录通常会包含更多条目。你只要在自己的工作站上键入 *ls -al /usr* 这个命令就能够展示这样的简单范例。你将会发现如 *man*、*src* 和 *local* 等这样的目录。FHS 中有一节专门探讨此目录的布局。然而，以大多数嵌入式 Linux 系统的用途来说，上面所建立的这三个目录已经够用了。

最后我们可以建立 */var* 的目录结构：

```
$ mkdir var/lib var/lock var/log var/run var/tmp
$ chmod 1777 var/tmp
```

同样地，此目录通常会包含更多的条目。尽管 *cache*、*mail* 和 *spool* 等目录对工作站或服务器来说很有用，但是嵌入式系统多半不需要这些目录。我们建立的目录必须符合“能够让嵌入式 Linux 系统上可以找到的大多数应用程序正常运行”这个最起码的要求。当然，如果需要 Web 服务或打印功能，那么你或许会想要加入额外的目录，以符合应用程序提供此功能的需要。FHS 和应用程序提供的文档可以找到实际需求。

准备好根文件系统的骨架之后，让我们将各种软件组件安装到正确的位置上去。

以不同的根文件系统结构来执行 Linux

正如之前的讨论，可以在 FHS 中找到建立根文件系统的规则。尽管大多数的 Linux 应用和发行套件都会遵照这些规则，但是 Linux 内核本身并未强制实施这些规则。事实上，内核源码与根文件系统结构的几乎没有什么关联。它让你可以使用截然不同的根文件系统结构来建立嵌入 Linux 系统。然后你必须修改大多数软件套件的默认值，以便与新的结构相符。有些人甚至采取更极端的做法，建立完全没有根文件系统的嵌入式 Linux 系统。不用说，我会建议你别这么做。以上描述的根文件系统的规则，是开放源码和自由软件所有开发者一致奉行的原则。如果使用其他规则来建立嵌入式 Linux 系统，无疑将自己排斥在大多数的开放源码和自由软件套件以及它们的开发者之外。

链接库

我们已经讨论过如何建立、安装和使用 GNU C 链接库以及应用程序开发时使用的链接库替代品。现在我们将讨论，如何在目标板的根文件系统上安装这些相同的链接库，让我们开发的应用程序使用他们。我们将会讨论 `diet libc`，因为它主要用作静态链接库。

`glibc`

正如我前文所说，`glibc` 套件包含若干链接库。你可以在套件建立期间列出 `$(TARGET_PREFIX)/lib` 目录的内容检查它所安装的所有链接库。此目录主要包含四种类型的文件：

实际的共享链接库

这类文件的文件名格式为 `libLIBRARY_NAME-GLIBC_VERSION.so`，其中 `LIBRARY_NAME` 是链接库的名称，`GLIBC_VERSION` 是你使用的 `glibc` 套件的版本编号。例如，`glibc 2.2.3` 的数学链接库的名称为 `libm-2.2.3.so`。

主修订版本的符号链接

主修订版本的编号方式与实际的 `glibc` 版号不同。以 `glibc 2.2.3` 实际的共享 C 链接库 `libc-2.2.3.so` 为例，它的主修订版本编号为 6。相对而言，`libdl-2.2.3.so` 的主修订版本编号为 2。主修订版本的符号链接的名称格式为 `libLIBRARY_NAME.so.MAJOR_REVISION_VERSION`，其中 `MAJOR_REVISION_VERSION` 是链接库的主修订版本编号。以实际的 C 链接库为例，其符号链接的名称为 `libc.so.6`。`libdl` 则是 `libdl.so.2`。程序一旦链接了特定的链接库，它将会参用其符号链接。程序启动时，加载器在加载程序之前，会因此加载该文件。

与版本无关的符号链接指向主修订版本的符号链接

这些符号链接的主要功能，是为需要链接特定链接库的所有程序提供一个通用的条目，与主修订版本的编号或 `glibc` 涉及的版本无关。这些符号链接典型的格式为 `libLIBRARY_NAME.so`。例如，`libm.so` 指向 `libm.so.6`，`libm.so.6` 指向实际的共享链接库 `libm-2.2.3.so`。唯一的例外是 `libc.so`，正如我在第四章所说，它是一个链接命令行。这个与版本无关之符号链接，就是链接程序时参用到的一个文件。

静态链接库包文件

选择以静态方式链接链接库的应用程序便会使用这些包文件。这些包的文件名格式为 `libLIBRARY_NAME.a`。例如 `libdl` 的静态包文件就是 `libdl.a`。

你还会在 `$(TARGET_PREFIX)/lib` 目录中找到若干其他类型的文件，例如 `crti.o` 和 `crt1.o`，不过你不必将这些文件复制到目标板的根文件系统。

以上描述的四种类型的文件，我们只需其中两种：实际的共享链接库和主修订版本的符号链接。其余两种类型的文件只有在链接执行文件的时候才会用到，执行应用程序的时候并不需要。

除了链接库文件，我们还需要复制动态链接器及其符号链接。动态链接器的文件名，依据 glibc 链接库的命名习惯，通常会叫作 *ld-GLIBC_VERSION.so*。然而，这或许是 GNU 工具链中最奇特的地方，动态链接器符号链接的名称取决于工具链所应用的架构。如果这是为 i386、ARM、SuperH 或 m68k 建立的工具链，则动态链接器的符号链接通常会叫作 *ld-linux.so.MAJOR_REVISION_VERSION*。如果这是为 MIPS 或 PowerPC 所建立的工具链，则动态链接器的符号链接通常会叫作 *ld.so.MAJOR_REVISION_VERSION*。

然而，在向目标板的根文件系统实际复制任何 glibc 组件之前，必须先找出应用程序需要哪些 glibc 组件。表 6-2 提供了 glibc 中所有组件的简短说明（注 1），以及每个组件的引用提示。除了我的提示，还需要根据程序的链接状态来评估程序需要哪些组件。

表 6-2: glibc 里的链接库组件以及根文件系统的引用提示

链接库组件	内容	引用提示
<i>ld</i>	动态链接器 ^a	必要
<i>libBrokenLocale</i>	修正进程，让 locale（本地）特性有问题的应用程序得以正常执行。 经由预先加载来覆盖应用程序的预设值（需使用 LD_PRELOAD。）	很少用到
<i>libSegFault</i>	用来捕捉内存区段错误（segmentation fault）以及进行回溯的进程	很少用到
<i>libanl</i>	异步名称查询进程	很少用到
<i>libc</i>	主 C 链接库进程	必要
<i>libcrypt</i>	密码学进程	大多数涉及认证之应用程序需要用到
<i>libdl</i>	用来动态加载共享目的文件的进程	使用 dlopen() 之类函数的应用程序需要用到
<i>libm</i>	数学进程	数学函数需要用到
<i>libmemusage</i>	用来进行堆（heap）和堆栈（stack）内存统计的进程	很少用到

注 1: 完整的说明参见 glibc 的使用手册。

表 6-2: glibc 里的链接库组件以及根文件系统的引用提示 (续)

链接库组件	内容	引用提示
<i>libnsl</i>	NIS 网络服务链接库进程	很少用到
<i>libnss_compat</i>	这是 NIS 与 Name Switch Service (NSS) 兼容的进程	由 glibc NSS 自动加载 ^b
<i>libnss_dns</i>	DNS 的 NSS 进程	由 glibc NSS 自动加载
<i>libnss_files</i>	文件查询的 NSS 进程	由 glibc NSS 自动加载
<i>libnss_hesiod</i>	Hesiod 名称服务的 NSS 进程	由 glibc NSS 自动加载
<i>libnss_nis</i>	NIS 的 NSS 进程	由 glibc NSS 自动加载
<i>libnss_nisplus</i>	NIS plus 的 NSS 进程	由 glibc NSS 自动加载
<i>libpcprofile</i>	程序计数器统计进程	很少用到
<i>libpthread</i>	Linux 的 Posix 1003.1c 多线程	多线程设计需要用到
<i>libresolv</i>	名称解析器进程	名称解析需要用到
<i>Librt</i>	异步 I/O 进程	很少用到
<i>libthread_db</i>	多线程调试进程	对使用多线程的应用程序进行调试时, 由 <i>gdb</i> 自动加载。事实上, 任何应用程序都不会链接此链接库
<i>Libutil</i>	登录进程, 它是用户记账数据库的一部分	终端联机管理需要用到

- a. 此链接库组件本身并非链接库。事实上, *ld.so* 是个由 ELF 二进制文件格式加载器所调用的可执行文件, 用来将动态链接库加载到应用程序的内存空间。
- b. 细节参见第四章。

除了记下应用程序链接哪些链接库, 通常还可以使用 *ldd* 命令列出应用程序要依存哪些动态链接库。然而在跨平台开发环境中, 主机的 *ldd* 命令无法处理目标板的二进制文件。不过你仍旧可以使用我们在第四章安装的跨平台 *readelf* 命令来找出应用程序依存哪些动态链接库。以下示范如何使用 *readelf* 取回 BusyBox 工具程序的依存关系:

```
$ powerpc-linux-readelf -a ${PRJROOT}/rootfs/bin/busybox | \
> grep "Shared library"
0x00000001 (NEEDED)                               Shared library: [libc.so.0]
```

然而, 如果安装了 uClibc, 应该使用 uClibc 安装的具有跨平台能力的 *ldd* 命令。对我们的控制模块目标板 (一个基于 PowerPC 的板子) 来说, 此命令的名称为 *powerpc-uclibc-*

ldd。通过它，可以列出目标板二进制文件依存哪些链接库。例如下面是 BusyBox 工具的依存关系（因为版面的关系，有一行的结尾被折到下一行）：

```
$ powerpc-uclibc-ldd ${PRJROOT}/rootfs/bin/busybox
libc.so.0=>/home/karim/control-project/control-module/tools/uclibc/lib/
libc.so.0
/lib/ld-uClibc.so.0 => /lib/ld-uClibc.so.0
```

决定需要哪些链接库组件之后，我们可以将这些链接库组件和相关的符号链接复制到目标板根文件系统中的 */lib* 目录里。下面便是一组用来复制必要 glibc 组件的命令：

```
$ cd ${TARGET_PREFIX}/lib
$ for file in libc libcrypt libdl libm \
> libpthread libresolv libutil
> do
> cp $file-*.so ${PRJROOT}/rootfs/lib
> cp -d $file.so.[*0-9] ${PRJROOT}/rootfs/lib
> done
$ cp -d ld*.so* ${PRJROOT}/rootfs/lib
```

第一个 *cp* 命令会复制实际的共享链接库，第二个 *cp* 命令会复制主修订版本的符号链接，第三个 *cp* 命令则会复制动态链接器及其符号链接。以上这三道命令是基于前文在这一节描述的 *\${TARGET_PREFIX}/lib* 目录中各个文件的命令规则。在第二个和第三个 *cp* 命令中的 *-d* 选项用来复制符号链接本身。否则会变成复制符号链接指向的文件。

当然，在以上列出的命令中，可以移除应用程序并未用到的链接库。如果想让根文件系统包含 glibc 的完整链接库，则可以使用如下的一组命令：

```
$ cd ${TARGET_PREFIX}/lib
$ cp *-*.so ${PRJROOT}/rootfs/lib
$ cp -d *.so.[*0-9] ${PRJROOT}/rootfs/lib
$ cp libSegFault.so libmemusage.so libpcprocfile.so \
> ${PRJROOT}/rootfs/lib
```

如果应用程序用到 glibc NSS，别忘了将你需要用到的 *libnss_SERVICE* 链接库复制到目标板的根文件系统。*libnss_files* 和 *libnss_dns* 是其中最常用的链接库。你还需要将 glibc 提供的 *nsswitch.conf* 范例文件复制到目标板的 */etc* 目录，并且根据设置对它进行定制（注 2）：

```
$ cp ${PRJROOT}/build-tools/glibc-2.2.1/nss/nsswitch.conf \
> ${PRJROOT}/rootfs/etc
```

注 2: *nsswitch.conf* 配置文件的定制细节可参考《Linux Network Administrator's Guide》（O'Reilly）。

不管你复制的是全部或部分的 glibc 链接库，将会发现其中不乏大型的链接库。想要缩减所安装链接库的尺寸，可以使用前面所建立的跨平台 strip 工具程序。切勿以 strip 来处理原始的链接库，因为你可能还需要用到它们。应该在链接库被复制到根文件系统后，再使用 strip 来处理它们：

```
$ powerpc-linux-strip ${PRJROOT}/rootfs/lib/*.so
```

以我的控制模块为例，将所有 glibc 组件复制到 `${PRJROOT}/rootfs/lib` 目录后，进行 strip 处理之前，所有组件约使用了 10 MB 的空间，进行 strip 处理之后该目录占用的空间缩减为 2.5 MB。

glibc 组件被安装到目标板的根文件系统之后，应用程序就可以在运行时使用它们。

uClibc

如同 glibc 一样，uClibc 也包含了若干链接库。你可以在 `${PREFIX}/uClibc/lib` 目录中看到完整的清单。如同 glibc 目录，此目录也包含四种不同类型的文件。

因为 uClibc 是 glibc 的替代品，所以 uClibc 中各个组件的名称及其用法如同 glibc 中相应的组件。因此，可以将表 6-2 应用在 uClibc 中相应的组件上。然而请注意，uClibc 并不会实现 glibc 中的所有组件。uClibc 只会实现 *ld*、*libc*、*libcrypt*、*libdl*、*libm*、*libpthread*、*libresolv* 和 *libutil*。你可以使用前一节针对 glibc 提到的方法，找出目标板需要哪些 uClibc 组件。

决定好所需要的组件清单后，接着可以将它们及相关的符号链接复制到目标板根文件系统中的 */lib* 目录里。下面便是一组用来复制必要 uClibc 组件的命令：

```
$ cd ${PREFIX}/uClibc/lib
$ for file in libuClibc ld-uClibc libc libdl \
> libcrypt libm libresolv libutil
> do
> cp $file-*.so ${PRJROOT}/rootfs/lib
> cp -d $file.so.[*0-9] ${PRJROOT}/rootfs/lib
> done
```

以上命令有可能会报告找不到两个文件：

```
cp: libuClibc.so.[*0-9]: No such file or directory
cp: libc-*.so: No such file or directory
```

这不是问题，因为这些文件原本就不存在。为了方便键入，上面这组命令相当精简，如果不想看到任何错误信息，请自行为每个 *cp* 命令加上条件语句。

如同 glibc 一样，可以根据自己的需求修改链接库复制清单。请注意，与 glibc 相比，在只复制 uClibc 组件的情况下，节省不了多少存储空间。以我的控制模块为例，将所有 uClibc 组件复制到根文件系统的 */lib* 目录后，占用了大约 300 KB 的空间。如下的命令会将 uClibc 的所有组件复制到目标板的根文件系统：

```
$ cd ${PREFIX}/uClibc/lib
$ cp *-*.so ${PRJROOT}/rootfs/lib
$ cp -d *.so.[*0-9] ${PRJROOT}/rootfs/lib
```

你不需要用 strip 处理 uClibc 组件，因为它们已经被 uClibc 自己的建立命令行 strip 过了。你可以使用 *file* 命令验证看看。

内核模块

我们曾在第五章建立内核模块，并且将它们安装到临时目录 *\${PRJROOT}/images*。现在我们可以将这些模块复制到它们在目标板的 */lib* 目录里的最终目的地。

因为你可能已经编译好若干内核版本准备测试目标板，所以需要你从中选出一组内核模块以便复制到根文件系统。以控制模块为例，我为目标板选择了 2.4.18 版的内核，并用如下的命令将内核的模块目录整个复制到根文件系统：

```
$ cp -a ${PRJROOT}/images/modules-2.4.18/* ${PRJROOT}/rootfs
```

此处使用 *cp* 命令的 *-a*（archive 模式）选项来复制文件和目录。这么做可以保存文件属性和链接以及递归地复制目录。请注意，在如上的命令中，并不需要特别为 *\${PRJROOT}/rootfs* 附上 */lib/modules* 路径，这与我们在第五章将模块安装到 *\${PRJROOT}/images/modules-2.4.18* 方式有关。

为目标板准备好内核模块后，可能还会想为它加入 */etc/modules.conf* 配置文件，以便在系统运行期间自动加载模块。至于模块管理以及使用 */etc/modules.conf* 配置文件的进一步细节请参考《Linux Device Drivers》第十一章的说明。

内核映像

正如前文所述，根文件系统上是否会出现实际的内核映像，这与引导加载程序的能力有非常大的关系。如果准备将引导加载程序设置成从根文件系统来启动内核，则此刻可以将内核映像复制到目标板的根文件系统：

```
$ mkdir ${PRJROOT}/rootfs/boot
$ cd ${PRJROOT}/images
$ cp zImage-2.4.18 ${PRJROOT}/rootfs/boot
```

除了内核映像、可能还想以它为典范复制用来建立该内核映像的配置文件，这样即使最初的项目工作空间不复存在，对你仍不会有丝毫的影响：

```
$ cp 2.4.18.config ${PRJROOT}/rootfs/boot
```

因为我们将在第九章讨论引导加载程序的设置，所以此处不再进一步探讨关于内核设置的问题。我们稍后再继续说明内核映像的设置。

设备文件

依照Unix的传统，在Linux系统中任何对象（包括设备）都可视为文件（注3）。在Linux根文件系统中，所有设备文件（亦称设备节点）都放在/dev目录里。对各种可能的系统变体来说，大多数的工作站和服务器发行套件为/dev目录附带了内容，这个数目超过了2000套。因为嵌入式Linux系统是个定制的系统，所以目标板的/dev目录里并不需要像Linux工作站和服务器那样填入那么多条目。事实上，只需建立让系统正常操作的必要条目即可。

如果手上没有可供参考的信息，很难判断自己需要哪些条目。倘若你选择用devfs（设备文件系统）来取代固定的静态设备文件，则可免去寻找设备信息的麻烦。然而，devfs并没有得到广泛采用，静态的设备文件仍是标准。

内核源码树的Documentation/devices.txt文档就是静态设备主要和次要编号的正式信息来源。每当你无法确定某个设备的名称或编号时请查阅此文档。

表6-3列示了/dev目录中需要填入的最基本条目。根据设置情况，或许应该加入若干额外的条目。在某些情况下，可能甚至需要使用表6-3以外的条目。例如在某些系统上第一个串行端口并非ttyS0。举例来说，SuperH-based系统的第一个串行端口是ttySC0（主要编号：204，次要编号：8），StrongARM-based系统的第一个串行端口则是ttySA0（主要编号：204，次要编号：5）。

表 6-3: 基本的/dev 条目

文件名	说明	类型	主编号	次编号	权限位
mem	物理内存存取	字符	1	1	600
null	null（黑洞）设备（译注1）	字符	1	3	666

注3： 显著的例外是网络接口（如Ethernet网卡）因为它们没有设备文件。

译注1： 进入此设备的任何数据皆会化为无形。

表 6-3: 基本的 /dev 条目 (续)

文件名	说明	类型	主编号	次编号	权限位
<i>zero</i>	以 null byte (零值字节) 为数据来源	字符	1	5	666
<i>random</i>	真随机数产生器	字符	1	8	644
<i>tty0</i>	现行的虚拟控制台	字符	4	0	600
<i>tty1</i>	第一个虚拟控制台	字符	4	1	600
<i>ttyS0</i>	第一个 UART 串行端口	字符	4	64	600
<i>ty</i>	现行的 TTY 设备	字符	5	0	666
<i>console</i>	系统控制台	字符	5	1	600

想知道建立设备文件的方法可参考《Running Linux》第六章的说明。基本上, 要使用 *mknod* 命令来建立每个条目。与本书到目前为止所使用的其他命令相比, 此处提到的命令需要你以 root 用户身份来执行。一旦建立好设备文件之后, 别忘了注销 root 用户。

以下示范如何建立表 6-3 中头几个条目:

```
$ cd $(PRJROOT)/rootfs/dev
$ su -m
Password:
# mknod -m 600 mem c 1 1
# mknod -m 666 null c 1 3
# mknod -m 666 zero c 1 5
# mknod -m 644 random c 1 8
...
# exit
```

除了基本的设备文件, */dev* 目录中还必须包含若干必要的符号链接。表 6-4 列示的便是此类符号链接。如同其他符号链接, 可以使用 *ln -s* 命令来建立这些链接。

表 6-4: 必要的 /dev 符号链接

链接名称	链接对象
<i>fd</i>	<i>/proc/self/fd</i>
<i>stdin</i>	<i>fd/0</i>
<i>stdout</i>	<i>fd/1</i>
<i>stderr</i>	<i>fd/2</i>

现在我们已经为目标板准备好了基本的 */dev* 目录。稍后我们将会回到该目录为某些类型的存储设备建立额外的条目。与设备文件和设备驱动程序有关的更完整探讨可参阅《Linux Device Drivers》一书。

自动建立 /dev 条目

Erik Andersen 扩展了某些文件系统（例如 JFFS2 和 CRAMFS）的建立工具，让它们使用设备表文件快速建立 /dev 条目。有了这个文件，就不需要以 root 身份用 `mknod` 命令来为目标板的根文件系统建立 /dev 条目。文件建立工具会剖析设备表文件并且建立相应的条目，就像它建立文件系统其余的部分一样，并不需要以 root 身份进行此工作。对 JFFS2 设备清单文件的支持已经成为 MTD 工具套件的一部分，并将此功能包含在 `mkfs.jffs2` 命令中。对 CRAMFS 设备清单文件的支持则是以 CRAMFS 源码套件的补丁形式提供的。该补丁以及最新版的 CRAMFS 文件系统建立工具可以从 <http://sourceforge.net/projects/cramfs/> 获得。我不会说明设备表文件的使用细节，因为在写作本书时能够使用设备表文件的 Linux 文件系统还十分有限。事实上，设备表文件的格式和使用，在 MTD 工具套件及 CRAMFS 补丁提供的 `device_table.txt` 文件里已经有相当好的说明。我们将会在第七章讨论 MTD 工具套件，并在第八章讨论 JFFS2 和 CRAMFS 文件系统。

主要的系统应用程序

除了内核的功能以及根文件系统的结构，Linux 还继承了 Unix 极其丰富的命令集。问题在于标准的工作站或服务器发行套件都配备有数以千计的二进制命令文件，而且不同发行套件提供的命令集还各不相同。显然，开发者没有能力逐一交叉编译这么多二进制文件，而且大多数嵌入式系统也不需要这么多二进制文件。

因此有两种做法：不是只挑选若干标准命令，就是尽可能将命令集浓缩成仅仅实现必要功能的极少数应用程序。接下来，我首先会讨论第一种做法。然而我并不建议采取这种做法，因为它相当冗长乏味。我会将重点摆在第二种做法，而且有各种计划实现它。尤其是，我将会探讨 BusyBox、TinyLogin 和 Embutils 这三个作为此用途的主要套件。

完整的标准应用程序

如果想有选择地利用主流发行套件中可以找到的若干标准应用程序，最好的对策就是以 Linux From Scratch 计划位于 <http://www.linuxfromscratch.org/> 的网站为起点。此计划提供有各种套件的说明和链接，其目的在于协助你定制自己的发行套件。该网站提供的《Linux From Scratch》便是此计划的主要文档，它逐一列出了每个应用程序的建立提示以及相关链接，并且为每个套件提供了建立时间和磁盘空间的估算值。

或者，可以从网络逐一下载应用程序并且依照每个套件的指示进行编译和交叉编译。因

为只有少数套件提供有交叉编译的完整说明，所以你可能需要检查套件的 Makefile，并据此决定适当的建立标志或对套件进行适当的修改，让交叉编译顺利完成。

BusyBox

Bruce Perens 于 1996 年发起的 BusyBox 计划，目的在于协助 Debian 发行套件建立安装磁盘。从 1999 年开始，此计划由 uClibc 的维护者 Erik Andersen 接手维护，它最初是 Lineo 开放源码成果的一部分，目前是个与厂商无关的计划。其间，BusyBox 计划发展很快，它现在是许多嵌入式 Linux 系统的基石之一。它被收纳在大多数嵌入式 Linux 发行套件之中，并且拥有非常活跃的用户社群。此计划的网站目前位于 <http://www.busybox.net/>。此计划网站包括了文档、链接以及以往的邮件清单记录。如果愿意接受 GNU GPL 的许可条款，可在同一个网站下载 BusyBox 套件。

BusyBox 之所以受到热烈的欢迎是因为它能够以一个极小型的应用程序来提供整个命令集的功能。BusyBox 实现了许多命令，以下列举一二：*ar*、*cat*、*chgrp*、*chmod*、*chown*、*chroot*、*cp*、*cpio*、*date*、*dd*、*df*、*dmesg*、*dos2unix*、*du*、*echo*、*env*、*expr*、*find*、*grep*、*gunzip*、*gzip*、*halt*、*id*、*ifconfig*、*init*、*insmod*、*kill*、*killall*、*ln*、*ls*、*lsmod*、*md5sum*、*mkdir*、*mknod*、*modprobe*、*more*、*mount*、*mv*、*ping*、*ps*、*pwd*、*reboot*、*renice*、*rm*、*rmdir*、*rmmmod*、*route*、*rpm2cpio*、*sed*、*stty*、*swapon*、*sync*、*syslogd*、*tail*、*tar*、*telnet*、*tftp*、*touch*、*traceroute*、*umount*、*uname*、*uuencode*、*vi*、*wc*、*which* 以及 *whoami*。

尽管 BusyBox 并不支持这些的命令的所有选项，它提供的子集已足以满足大多数应用的要求。你可以在 BusyBox 发行套件的 *docs* 目录里看到若干不同格式的文件。

BusyBox 支持第三章提到的所有架构。它可以用静态和动态的方式链接 glibc 或 uClibc。你还可以修改 BusyBox 缺省的建立配置，移除不想使用的命令。

设置

首先，需要从该计划的网站将 BusyBox 套件的副本下载到 `${PRJROOT}/sysapps` 目录。以我的控制模块为例，我将会使用 BusyBox 0.60.5。

将套件解开之后，我们可以移往这个目录，以便进行设置的其余步骤：

```
$ cd ${PRJROOT}/sysapps/busybox-0.60.5
```

尽管此套件的 CVS 版本，如同我在第四章提到的 uClibc 一样，包括可以用来设定配置选项的终端方式的菜单工具，然而此套件的稳定版本，例如我使用的套件，必须通过编辑 *Config.h* 文件的方式来设定配置选项。此文件用 C 语言的 `#define` 语句来设定每个选项。你可以使用 `//`（双斜线）将选项的 `#define`（定义）注释掉来停用相应的选项。

有两种选项可供设定：命令支持选项以及功能支持选项。停用或启用命令支持选项可以移除或加入相应的命令。把 `#define BB_MKNOD` 这一行设定改为 `//#define BB_MKNOD`，可以停止 BusyBox 对 `mknod` 命令的支持。功能支持选项具有类似的作用。然而，这些功能未必与特定命令有任何关系。每个功能支持选项 (`#define BB_FEATURE_...`) 之前都会有一行注释对相应的功能进行说明。

务必查明缺省启用的命令支持哪些选项。某些重要的命令，例如 `ifconfig`、`insmod` 和 `ping`，缺省是禁用的。

除了配置文件，主要的 Makefile 包含了若干标志，可用来控制 BusyBox 的建立方式。这些标志主要是在 BusyBox 开发期间进行调试时用到，而且标准发行套件不会用到他们。你惟一可能想要修改的标志就是 `DOSTATIC`。如果将此标志设为 `true`，BusyBox 二进制文件会被静态链接至 C 链接库。`DOSTATIC` 的缺省值为 `false`，这会导致链接二进制文件采用动态链接方式。要改变该标志，可以修改 Makefile，或者用 `CDOSTATIC=true` 参数来执行 `make` 命令。

设定好 BusyBox 后，接着进行编译和安装的设置。欲链接 `glibc`，可使用如下的命令：

```
$ make TARGET_ARCH=ppc CROSS=powerpc-linux- \
> PREFIX=${PRJROOT}/rootfs all install
```

`TARGET_ARCH` 变量让 Makefile 判断是否可以进行某些架构无关的优化。如同其他的环境，`CROSS` 变量可用来指定跨平台开发工具的前置名称。最后，`PREFIX` 变量则可用来设定根文件系统的基础目录。Makefile 会把所有的 BusyBox 组件安装到此目录。

如果要用 `uClibc`（替代 GNU C 链接库）来建立 BusyBox，可使用如下命令：

```
$ make TARGET_ARCH=ppc CROSS=powerpc-uclic- \
> PREFIX=${PRJROOT}/rootfs all install
```

BusyBox 现在已经安装到目标板的根文件系统，接下来说明它的用法。

用法

为了充分了解 BusyBox 的使用方法，让我们先来看看 BusyBox 建立程序在目标板根文件系统上安装的组件。正如预料的那样，它只安装了一个二进制文件 `/bin/busybox`。此二进制文件支持 `Config.h` 配置的命令。该二进制文件并不会被直接调用，而是通过指向 `/bin/busybox` 的符号链接（以原始命令的名称为名）间接调用此二进制文件。这些符号链接将会在可以找到执行文件的所有目录中建立，包括 `/bin`、`/sbin`、`/usr/bin` 和 `/usr/sbin`。当你在系统正常运行期间键入某一命令，将会通过符号链接调用 `busybox` 命令。接着，`busybox` 会根据调用它的命令名称来判断你想要执行的命令。例如，`/bin/l` 是指向 `/bin/`

busybox 的符号链接。当你键入 *ls*，便会调用 *busybox* 命令，接着 *busybox* 会判定你想使用 *ls* 命令，因为 *ls* 是命令行上的第一个参数（注 4）。

尽管这个方法既简单又有效，但这也代表你无法随意为这些符号链接取名字。如果为 */bin/dir* 或 */bin/busybox* 建立了一个名为 */bin/dir* 符号链接，不会有任何作用，因为 *busybox* 并不认识 *dir* 命令。

请注意，虽然 *BusyBox* 通常会使用符号链接来为所替代的命令建立指向 */bin/busybox* 的链接，不过你也可以在 *BusyBox* 安装期间指示它建立硬链接。然而，它们在运行时的作用都是一样的，与使用的链接类型无关。

你可以在计划网站上提供的文档（套件中也附带）找到每个命令支持哪些选项。通常，*BusyBox* 支持的选项，其功能如同原始命令提供的选项。例如，对 *BusyBox* 的 *ls* 使用 *-al* 选项，其效果如同对原始的 *ls* 使用相同的选项。

当你使用 *BusyBox* 提供的 shell 时，例如 *ash*、*lash* 或 *msh*，可以使用 */etc/profile* 文件为所有的 shell 用户定义全局变量。下面是为单用户的目标板定义的 */etc/profile* 范例文件：

```
# Set path
PATH=/bin:/sbin:/usr/bin:/usr/sbin
export PATH
```

除了设定搜索路径，你还可以设定 *LD_LIBRARY_PATH* 环境变量，每个应用程序会在启动期间使用此变量来找到与其有依存关系的链接库。尽管链接库缺省的位置是 */lib*，不过系统可能会将链接库放在其他目录。如果真是这样的话，可以借着将适当的目录路径加入 *LD_LIBRARY_PATH* 变量，迫使动态链接器到其他位置寻找链接库。如同 *PATH* 环境变量，当你要将多个目录路径加入链接库路径时，必须以冒号（:）隔开每个目录路径。

请注意，在工作站或服务器上，通常只会使用 *LD_LIBRARY_PATH* 变量来暂时存放新的链接库路径。要永久加入另一个链接库路径，需要编辑 */etc/ld.so.conf* 配置文件。接着需要使用 *ldconfig* 命令根据这个文件来产生 */etc/ld.so.cache*。动态链接器会依照 */etc/ld.so.cache* 文件的指示找到应用程序动态链接的链接库。*ldconfig* 是我们在第四章编译 *glibc* 时建立的，但是它是个目标板二进制文件，无法在主机上使用它来产生目标板的 *ld.so.cache*。

注 4：如同任何其他的应用程序，用来调用 *busybox* 的命令行会被传递给 *busybox* 的 *main()* 函数。

TinyLogin

与BusyBox非常类似，TinyLogin将许多登录工具程序放在单个二进制文件中。尽管能够单独使用TinyLogin，不过它通常会与BusyBox并用。这两个套件由相同的开发者维护，因此很容易一起使用。因为它们通常会一起使用，该计划的开发者已经将TinyLogin的所有功能整合进BusyBox的CVS版本，一旦CVS开发版作为稳定版发布，这两个套件就会合二为一。然而，会发现继续在BusyBox之外使用TinyLogin的功能还是有好处的。主要是因为，TinyLogin中实现的许多命令必须以root的特权来执行，因此TinyLogin二进制文件的拥有者必须是root用户，而且启用它的set user权限位——这种配置就是一般所谓的setuid root。因为TinyLogin使用符号链接的方式与BusyBox一样，所以一个包含这两个套件功能的二进制文件支持的命令，例如ls，也必须以root身份来执行，如果命令有编程（程序设计）上的错误，这会增加有心者据此获得root特权的可能性。尽管BusyBox会在不使用的时候放弃root特权，但是你也可以编辑配置文件将那些命令设定成需要root特权，因此使用独立套件提供的TinyLogin功能是最安全的设置。

你可以在TinyLogin计划位于<http://tinylogin.busybox.net/>的网站找到相关文档、以往的邮件论坛记录、其他网站的链接，以及用来下载TinyLogin套件的链接指示。以我的控制模块为例，我使用的是1.2版的TinyLogin。

如同BusyBox一样，TinyLogin支持第三章提到的所有架构，而且可以用静态或动态的方式链接glibc或uClibc。TinyLogin可以有效地取代以下命令：*addgroup*、*adduser*、*delgroup*、*deluser*、*getty*、*login*、*passwd*、*su*、*sulogin*以及*vlock*。

设置

安装TinyLogin的第一步就是将套件下载到`${PRJROOT}/sysapps`目录，将套件解开之后，我们可以移往它所建立的目录，以便进行设置的其余步骤：

```
$ cd ${PRJROOT}/sysapps/tinylogin-1.2
```

TinyLogin配置设定的方式非常像BusyBox，就是编辑`Config.h`配置文件，将不需要的命令支持选项及功能支持选项注释掉。TinyLogin的Makefile还具有与BusyBox类似的选项。前面对BusyBox的`Config.h`文件和Makefile所做的说明对TinyLogin同样适用。

除了需要设定的其他选项，请特别注意USE_SYSTEM_PWD_GRP和USE_SYSTEM_SHADOW这两个Makefile的选项。前面对Makefile里的语句所做的说明，应该能够让你认识到这些选项的作用。USE_SYSTEM_PWD_GRP通常应该设成false，除非你准备使用glibc的NSS链接库以及有适当设定的`/etc/nsswitch.conf`配置文件。如果将此选项设成false，

TinyLogin 将会直接使用 */etc/passwd* 和 */etc/group* 文件，而不会使用 glibc 提供的 password 和 group 功能。

同样地，如果将 `USE_SYSTEM_SHADOW` 设成 `false`，TinyLogin 将会使用自己的影子功能来存取影子密码。传统上，系统上的任何人都可以读取 */etc/passwd*，在密码破解程序越来越多的现在，这样很不安全。因此所谓的影子密码变成了标准。使用此机制时，*/etc/passwd* 文件中的密码字段只包含填充字符，加密后的密码存放在 */etc/shadow* 文件中，而且只有以 root 特权执行的进程才有权读取。请注意，如果之前将 uClibc 设成不支持影子文件，但是现在却将 `USE_SYSTEM_SHADOW` 设成 `true`，在链接 uClibc 的时候就会失败。

如同 BusyBox 一样，倘若你想用静态链接的方式来建立 TinyLogin，可以把 `DOSTATIC` 设成 `true`。

设好 TinyLogin 的配置之后，接着建立套件。（TinyLogin 的建立程序与 BusyBox 不同，无法在同一个步骤中编译和安装套件，必须先完成编译然后再进行安装，理由会在下面说明。）

欲以 glibc 来编译 TinyLogin，可使用如下的命令：

```
$ make CROSS=powerpc-linux- \
> PREFIX=${PRJROOT}/rootfs all
```

欲以 uClibc 来编译 TinyLogin，可使用如下的命令：

```
$ make CROSS=powerpc-uclibc- \
> PREFIX=${PRJROOT}/rootfs all
```

建立好套件之后，接着安装套件。因为安装程序必须对 TinyLogin 二进制文件设定 `setuid` 的权限，所以必须以 root 的身份来执行安装命令：

```
$ su -m
Password:
# make PREFIX=${PRJROOT}/rootfs install
# exit
```

TinyLogin 现在已经安装到目标板的根文件系统，接下来说明它的用法。

用法

TinyLogin 安装程序只会复制一个二进制文件（即 */bin/tinylogin*）到根文件系统。如同 BusyBox，以原始命令的名称为名的符号链接将会在各个可以找到二进制文件的目录中建立。

你将需要建立群组文件、密码文件和影子密码文件（分别为 */etc/group*、*/etc/passwd* 和 */etc/shadow*）以供各个 TinyLogin 命令使用。可惜，除非在目标板上执行 TinyLogin，否则 TinyLogin 无法建立这些文件。因此你必须复制并且手动编辑既有的文件，让它们能够在目标板的根文件系统上使用。一个简单的做法就是使用来自工作站的设置文件，同时为目标板上的用户保留相应的条目。通常最后只会剩下 root 用户。

你可以先将工作站上的群组和密码文件复制到目标板的 */etc* 目录，然后手动编辑目标板上的副本，移除目标板上不会用到的条目。然而影子文件需要多花一点功夫，因为你可能不希望将自己工作站的密码泄露给嵌入式系统的用户知道。要在目标板的影子文件中建立有效的条目，最简单的做法就是在工作站上建立假的用户，为这些用户设定密码，然后复制所产生的条目。下面是我的工作站上一个假用户 tmp 的条目：

```
tmp:$1$3cd0SElf$XWRL0KIL7vMSfLYbRCWaf/:11880:0:99999:7:-1:-1:0
```

为了方便起见，我会将这个用户的密码设成“root”。接着我会将此条目复制到目标板的影子文件，并且编辑用户名称：

```
root:$1$3cd0SElf$XWRL0KIL7vMSfLYbRCWaf/:11880:0:99999:7:-1:-1:0
```

我的目标板上现在有一个密码为“root”并且名称为“root”的用户。

不要忘了，密码文件会为每个用户指定它所使用的 shell 的名称。因为 BusyBox 提供的 shell 的命令名称是 *sh*，并且因为大多数工作站上的 shell 缺省是 *bash*，所以你需要将此名称改成目标板上的 shell。下面是 root 用户的密码文件条目：

```
root:x:0:0:root:/root:/bin/sh
```

TinyLogin 会将每个用户的搜索路径设为：

```
PATH=/bin:/usr/bin
```

如果想要改变该默认值，可以建立全局的 */etc/profile* 文件，正如前文所述，或是在每个用户的 home 目录中建立 *.profile* 文件。你将会发现如下的 *.profile* 文件对 root 用户很有用：

```
PATH=/bin:/sbin:/usr/bin:/usr/sbin
export PATH
```

欲进一步了解群组文件、密码文件、影子密码文件以及一般的系统管理，可参考 LDP 的《Linux System Administrator's Guide》、O'Reilly 的《Running Linux》以及前面所提到的《Linux From Scratch》。

embutils

embutils是针对主流Unix命令提供的一组经过简化和优化的替代品。embutils的撰写者和维护者是diet libc的作者Felix von Leitner，它的目标与diet libc非常类似。目前embutils支持第三章所提到的四种架构：ARM、i386、PPC和MIPS。embutils可以从<http://www.fefe.de/embutils/>获得（注5）。

尽管embutils会将若干命令放在单个二进制文件中，但是它主要想做到的是为每个命令提供小型的二进制文件。embutils提供如下的命令：*arch*、*basename*、*cat*、*chmgrp*、*chmod*、*chown*、*chroot*、*chvt*、*clear*、*cp*、*dd*、*df*、*dirname*、*dmesg*、*domainname*、*du*、*echo*、*env*、*false*、*head*、*hostname*、*id*、*install*、*kill*、*ln*、*ls*、*md5sum*、*mesg*、*mkdir*、*mkfifo*、*mknod*、*mv*、*pwd*、*rm*、*rmdir*、*sleep*、*sleep2*、*soscp*、*sosln*、*soslns*、*sosmv*、*sosrm*、*sync*、*tail*、*tar*、*tee*、*touch*、*tr*、*true*、*tty*、*uname*、*uniq*、*wc*、*which*、*whoami*、*write*以及*yes*。

如同BusyBox，这些替代品并不支持原始命令提供的所有选项，不过足以满足大多数系统操作的要求。然而跟BusyBox不同的是，embutils只能静态链接diet lib。它不能链接任何其他的链接库。因为diet libc已经很小了，所以链接后所产生的二进制文件当然也不会大。然而就整体的大小来看，BusyBox和embutils差不多。

设置

开始设置之前，需要先将diet libc安装在主机系统上（参见第四章的说明）。现在下载embutils并将它解开放到\${PRJROOT}/sysapps目录。我的控制模块使用的是embutils 0.15。接着可以移往套件的源码目录，以便进行设置的其余步骤：

```
$ cd ${PRJROOT}/sysapps/embutils-0.15
```

embutils没有配置设定能力，因此你可以立即建立套件：

```
$ make ARCH=ppc CROSS=powerpc-linux- all
```

然后你可以安装embutils：

```
$ make ARCH=ppc DESTDIR=${PRJROOT}/rootfs prefix="" install
```

建立和安装embutils过程中使用的选项和变量具有与diet libc相同的意义。

注5：如同diet libc，最后一个斜线（/）很重要。

用法

embutils 安装程序会将若干静态链接的二进制文件复制到目标板根文件系统的 */bin* 目录。与 BusyBox 不同的是，这些二进制文件只会安装到该目录。

如同 BusyBox，embutils 安装程序也会安装包罗所有的单个二进制文件。这个二进制文件的使用方式如同 BusyBox 一样，也是通过适当的符号链接。与 BusyBox 不同的是，需要手动建立这些符号链接，因为安装脚本不会自动建立这些链接。这个大的二进制文件支持以下命令：*arch*、*basename*、*chvt*、*clear*、*dmesg*、*dirname*、*domainname*、*echo*、*env*、*false*、*hostname*、*pwd*、*sleep*、*sync*、*tee*、*true*、*tty*、*uname*、*which*、*whoami* 以及 *yes*。

定制应用程序

根文件系统中有多处可以用来存放自己的应用程序，要存放在哪里取决于所拥有的组件数量和类型。通常可以根据 FHS 的指示来放置软件。

如果应用程序包含相对较少的二进制文件，将它们放到 */bin* 目录或许是最好的选择。这是第四章的命令监控程序使用的实际安装路径。

如果应用程序包含了一组错综复杂的二进制文件，并且可能包括数据文件，可以考虑为他们在根文件系统中新增一个目录。你可以将这个新目录取名为 *project*，或是按照自己的计划来取名字。以我的控制模块为例，可以为这个目录取名为 *control-module*。

定制目录可以包含根据需求定制的目录结构。如果将二进制文件摆在定制目录，可能需要在目标板上设定 PATH 环境变量，以便纳入该目录的路径。

请注意，在根文件系统中新增定制目录有违 FHS 的原则。然而这对标准不会有什么妨害，因为你只是在为目标板定制文件系统，它不可能成为自己的发行套件。

系统初始化

系统初始化是 Unix 系统的另一个特性。正如第二章所说，内核最后一个初始化动作就是启动 *init* 程序。此程序在终结系统启动程序之前会衍生各种应用程序，并且启动若干关键的软件组件。大多数 Linux 系统使用的 *init* 跟 System V 的 *init* 相仿，配置设定的方式也极为相似。对嵌入式 Linux 系统来说，System V *init* 过于灵活，因为此类系统很少会被当成多用户系统来使用。

上面这道命令会将 `BIN_OWNER` 和 `BIN_GROUP` 变量设成当前用户。Makefile 在安装各种组件的时候，缺省情况下会把它们的拥有者设成 `root` 用户。因为你并未登录成 `root` 用户，所以 Makefile 将会执行失败。对目标板而言，二进制文件的拥有者是谁并没有多大的关系，因为它并非多用户系统。然而，如果它是个多用户系统，就必须登录成 `root` 用户，然后执行安装命令。但无论如何，为 `ROOT` 设值的时候要非常小心，务必将它指向目标板的根文件系统。否则工作站的 `init` 最后可能会被目标板的二进制文件覆盖掉。你也可以选择不要登录成 `root` 用户，先以一般用户的权限执行安装命令，然后再以 `root` 用户身份执行 `chown` 命令，为安装的每个文件修改所有权。然而你必须浏览 Makefile，找出所安装的每个文件以及安装位置。

欲使用安装在目标板根文件系统上的 `init`，需要加入适当的 `/etc/inittab` 文件，以及在 `/etc/rc.d` 目录中填入适当的文件。基本上，`/etc/inittab` 将会为系统定义运行级别，`/etc/rc.d` 目录中的文件则是用来定义每个运行级别将会执行哪些服务。表 6-5 列出了 `init` 的七个运行级别以及它们在工作站和服务器的发行套件中的典型用法。

表 6-5: System V `init` 运行级别

运行级别	说明
0	终止系统的运行
1	单一用户模式，不需要登录程序
2	多用户模式但不支持 NFS，使用命令行形式的登录程序
3	多用户模式，使用命令行形式的登录程序
4	不使用
5	X11，使用图形用户接口形式的登录程序
6	重新启动系统

每个运行级别都会拥有一组相应的应用程序。例如，当工作站进入运行级别 5 时，`init` 会启动 X11 并以图形模式的登录程序提示用户键入用户名和密码。当你切换运行级别的时候，先关闭原运行级别启动的服务，然后启动新运行级别的服务。在这个机制中，运行级别 0 和 6 具有特殊的意义。尤其是，它们可让你用来安全地停止系统的运行。这可能涉及了，例如卸载根文件系统之外的文件系统，重新以只读模式安装根文件系统，以免损坏文件系统。

在大多数工作站上，缺省情况下系统启动时会进入运行级别 5。对嵌入式系统而言，如果没有存取控制的需要，则可以将缺省的运行级别设成 1。系统启动后，欲改变系统的运行级别，可以使用 `init` 或 `telinit` 命令；`telinit` 是 `init` 的符号链接。不管你使用的是哪个

命令，新发布的 *init* 命令将会通过 */dev/initctl* 这个 fifo（译注 2）与原始的 *init* 进程通信。最后我需要在目标板的根文件系统中建立相应的条目：

```
$ mknod -m 600 ${PRJROOT}/rootfs/dev/initctl p
```

欲进一步了解 */etc/inittab* 的格式以及 */etc/rc.d* 目录中可以找到哪些文件，可参考前面所提到的资源。

BusyBox init

除了缺省支持的命令，BusyBox 还提供与 *init* 类似的能力。如同原始的主流 *init*，BusyBox 也可以处理系统的启动工作。BusyBox 的 *init* 尤其适合在嵌入式系统中使用，因为它可以为嵌入式系统提供所需要的大部分 *init* 功能，却不会让嵌入式系统被 System V *init* 的额外特性拖累。此外，因为 BusyBox 是单个套件，所以当你要开发或维护系统时，不需要注意额外的软件套件。然而有些时候系统可能不适合使用 BusyBox 的 *init*，例如它并不提供运行级别的支持。

因为我已经说明过如何获得、设定和建立 BusyBox，所以将会把此处的讨论局限在 *init* 配置文件的设置上。

因为 */sbin/init* 是 */bin/busybox* 的符号链接，所以 BusyBox 是目标板系统上执行的第一个应用程序。当 BusyBox 知道调用它的目的是要执行 *init*，它会立即跳转到 *init* 进程。

BusyBox 的 *init* 进程会依次进行以下工作：

1. 为 *init* 设置信号处理进程。
2. 初始化控制台。
3. 剖析 *inittab* 文件、*/etc/inittab* 文件。
4. 执行系统初始化命令行。BusyBox 在缺省情况下会使用 */etc/init.d/rcS* 命令行。
5. 执行所有会导致 *init* 暂停的 *inittab* 命令（动作类型：wait）。
6. 执行所有仅执行一次的 *inittab* 命令（动作类型：once）。

一旦完成以上工作，*init* 进程便会循环执行以下工作：

1. 执行所有终止时必须重新启动的 *inittab* 命令（动作类型：respawn）。
2. 执行所有终止时必须重新启动但启动前必须先询问用户的 *inittab* 命令（动作类型：askfirst）。

译注 2：fifo 又称为命名管道（named pipe），它是进程之间具先进先出特性的通信管道。

在控制台初始化期间, BusyBox 会判断系统是否被设成在串行端口上执行控制台(例如, 把 console=ttyS0 作为内核引导参数)。若是这样, BusyBox 在 0.60.4 之前的版本会停用所有的虚拟终端。但是, 从 0.60.4 的版本开始, BusyBox 在初始化的过程中并不会停用虚拟终端。如果事实上没有虚拟终端, 稍后若用户想在某些虚拟终端上启动 shell 则会导致执行失败, 所以不必停用虚拟终端。

控制台初始化后, BusyBox 会检查 `/etc/inittab` 文件是否存在。如果此文件不存在, BusyBox 会使用缺省的 inittab 配置。它主要会为系统重引导、系统挂起以及 `init` 重启动设置缺省的动作。此外, 它还会为头四个虚拟控制台 (`/dev/tty1` 到 `/dev/tty4`) 设置启动 shell 的动作。如果并未建立这些设备文件, BusyBox 将会报错。

如果存在 `/etc/inittab` 文件, BusyBox 会予以剖析, 并将其中的命令记录在内部的数据结构中, 以便适时执行。BusyBox 能够识别的 inittab 文件格式, 在 BusyBox 套件附带的文档中有很好的说明。BusyBox 套件附带的文档中还包含详尽的 inittab 文件范例。

inittab 文件中每一行的格式如下所示:

```
id:runlevel:action:process
```

尽管此格式与传统的 System V `init` 类似, 但请注意, `id` 在 BusyBox 的 `init` 中具有不同的意义。对 BusyBox 而言, `id` 用来指定所启动进程的控制 tty。如果所启动的进程并不是个可以交互的 shell, 大可以将此字段空着不填。可以交互的 shell, 例如 BusyBox 的 `sh`, 应该会有个控制 tty。如果控制 tty 不存在, BusyBox 的 `sh` 将会报错。BusyBox 将会完全忽略 `runlevel` 字段, 所以你可以将它空着不填。`process` 字段用来指定所执行程序的路径, 包括命令行选项。`action` 字段用来指定表 6-6 所列八个可应用到 `process` 的动作之一。

表 6-6: BusyBox init 能够识别的 inittab 动作类型

动作	结果
sysinit	为 <code>init</code> 提供初始化命令行的路径
respawn	每当相应的进程终止执行便重新启动
askfirst	类似 <code>respawn</code> , 不过它的主要用途是减少系统上执行的终端应用程序的数量。它将会促使 <code>init</code> 在控制台上显示 “Please press Enter to activate this console.” 的信息, 并在重新启动进程之前等待用户按下 Enter 键
wait	告诉 <code>init</code> 必须等到相应的进程完成后才能继续执行
once	仅执行相应的进程一次, 而且不会等待它完成
ctrlaltdel	当按下 Ctrl-Alt-Delete 组合键时, 执行相应的进程

表 6-6: BusyBox init 能够识别的 inittab 动作类型 (续)

动作	结果
shutdown	当系统关机时，执行相应的进程
restart	当 <i>init</i> 重新启动时，执行相应的进程。通常此处所执行的进程就是 <i>init</i> 本身

以下是我的控制模块的简单 inittab 文件：

```
::sysinit:/etc/init.d/rcS
::respawn:/sbin/getty 115200 ttyS0
::respawn:/control-module/bin/init
::restart:/sbin/init
::shutdown:/bin/umount -a -r
```

这个 inittab 文件将执行以下动作：

1. 将 */etc/init.d/rcS* 设成系统的初始化文件。
2. 在 115200 bps 的串行端口上启动一个登录会话。
3. 启动控制模块的定制软件初始化命令行。
4. 如果 *init* 重新启动，将 */sbin/init* 设成它会执行的程序。
5. 告诉 *init*，它可以在系统关机的时候执行 *umount* 命令卸载所有文件系统，并且在卸载失败时用只读模式重新安装以保护文件系统。

此处将 *id* 空着不填，因为该字段与命令的正常操作无关。*runlevel* 也空着未填，因为 BusyBox 会完全忽略此字段。

然而，如同前文所述，除非 *init* 执行系统初始化命令行，否则不会进行以上提到的这些动作。这个命令行可能相当复杂，也可能会调用其他的命令行。你可以使用这个命令行来设定所有的基本设定值，初始化各种需要特别处理的系统组件。尤其是，此处是执行以下动作的好地方：

- 以读写模式重新安装根文件系统。
- 安装额外的文件系统。
- 初始化及启动网络接口。
- 启动系统监控程序。

下面是我的控制模块的初始化命令行：

```
#!/bin/sh
```

```
# 以读写模式重新安装根文件系统（需要 /etc/fstab）
mount -n -o remount,rw /

# 安装 /proc 文件系统
mount /proc

# 启动网络接口
/sbin/ifconfig eth0 192.168.172.10
```

上面这个初始化命令行要能够正常执行，目标板的根文件系统中必须存在 */etc/fstab* 文件。我将不会探讨这个文件的内容和用法，因为《Running Linux》已经有深入的讨论了。尽管如此，下面还是列出我的控制模块在开发期间使用的 */etc/fstab* 文件：

```
# /etc/fstab
# 设备          目录          类型      选项
#
/dev/nfs         /              nfs       defaults
none            /proc          proc      defaults
```

此例中，我通过 NFS 安装目标板的根文件系统以简化开发工作。我将会在第八章探讨文件系统的类型，以及在第九章探讨 NFS 的安装。

Minit

Minit 是 Felix von Leitner 开发的小型化工具（例如 *diet libc* 和 *embutils*）的一个组件。Minit 可以从 <http://www.fefe.de/minit/> 获得（注6）。如同 Felix 发行的其他工具一样，Minit 必须使用设定正确的 *diet libc*。

Minit 的初始化程序与传统的 System V *init* 完全无关。例如，Minit 并不会使用 */etc/inittab*，它所仰仗的是 */etc/minit* 目录结构的适当设定。你可以在 <http://www.fbunet.de/minit.shtml> 获得 Firdtjof Busse 对此（如何操作 Minit）撰写的文档，还可以在该处获得 */etc/minit* 目录的范例。

可惜，直到 0.8 版，Minit 仍不如 Felix 提供的其他工具那样成熟。例如，它的 Makefile 无法在非主机自己的根文件系统中安装各种组件。目前 Minit 并不适合嵌入式系统使用，不过在可预见的未来它将会值得一试。

注6： 如同可以从 *fefe.de* 获得的其他工具一样，最后一个斜线（/）很重要。



第七章

存储设备管理

嵌入式系统使用的存储设备通常跟工作站和服务器的有很大的差异。嵌入式系统倾向使用固态存储设备，例如 flash 芯片和 flash 磁盘。如同任何其他的 Linux 系统一样，使用这些设备需要对内核进行适当的设置和配置。因为这些存储设备跟典型的工作站和服务器的磁盘有很大的差异，所以用来操作它们（例如建立分区，复制文件，抹除）的工具当然也不会一样。这些工具便是本章要探讨的主题。

这一章，我们将讨论如何操作 Linux 使用的嵌入式存储设备。我首先会把重点放在 MTD 子系统支持的设备的操作上。我还会简述磁盘设备的操作。然而，如果想在系统中使用传统的磁盘设备，建议阅读讨论 Linux 系统维护的书籍，例如 O'Reilly 出版的《Running Linux》。本章最后一节将会提到交换功能在嵌入式系统中的使用情况。

MTD 支持的设备

正如我们在第三章“MTD”中所见，MTD 子系统非常复杂。要在目标板上使用它，不仅要适当配置内核，而且也要到该计划的网站取得 MTD 工具。接下来我们将会探讨这两个问题。

如同其他的内核子系统，MTD 子系统和 MTD 工具的发展独立于主流内核之外。因此，最新版的内核通常不会包含 MTD CVS 库里最新的程序代码。尽管如此，还是可以自己取回最新的程序代码，并用它取代你选用的内核中包含的 MTD 程序代码。

然而，因为主流内核版本中的 MTD 程序代码与 MTD 的发展并不同步，所以有时你可能会遇到问题。例如，我不能让一般的 Linux 2.4.18 内核从 DiskOnChip (DOC) 2000 引导，因为内核中缺省的 MTD 程序代码有缺陷。为了修正此问题，我根据 MTD 邮件论坛以往的记录找到的指示，自己手动修改 MTD 程序代码。因此，将会发现 MTD 邮件论坛以及它以往的记录很有用。

接下来，我首先会探讨 MTD 子系统的基本用法。这将会涵盖以下主题：配置内核、安装必要的工具，以及在 */dev* 目录中建立适当的条目。接着会将探讨的重点放在嵌入式 Linux 系统中最常使用的两种固态存储设备：与 CFI 兼容的原生 flash 设备以及 DOC 设备。

MTD 的基本用法

前面已经详细说明过了 MTD 子系统的架构，现在让我们专注在其组件的实际使用上。首先，我们将讨论 MTD 子系统需要的 */dev* 条目。其次，我们会讨论基本的 MTD 内核配置选项。然后，我们会讨论 Linux 中可用来操作 MTD 存储设备的工具。最后，我们将讨论如何在主机和目标板上安装这些工具。

MTD/dev 条目

有五种 MTD */dev* 条目和七种相应的 MTD 用户模块（注 1）。实际上，多个 MTD 用户模块可以共享相同的 */dev* 条目，并且每个 */dev* 条目可作为多个 MTD 用户模块的接口。表 7-1 描述了 MTD */dev* 条目的各种类型以及相应的 MTD 用户模块，表 7-2 则提供了次要编号的范围，并且描述了每种设备使用的命名方式。

表 7-1: MTD */dev* 条目、相应的 MTD 用户模块，以及设备主要编号

<i>/dev</i> 条目	可存取的可 MTD 用户模块	设备类型	主要编号
<i>mtdN</i>	字符设备	字符	90
<i>mtdrN</i>	字符设备（只读）	字符	90
<i>mtdblockN</i>	块设备、只读块设备、JFFS 以及 JFFS2	块	31
<i>nftlN</i>	NFTL	块	93
<i>ftlN</i>	FTL	块	44

表 7-2: MTD/dev 条目、次要编号，以及命名方式

<i>/dev</i> 条目	次要编号的范围	命名方式
<i>mtdN</i>	0 到 32，每次递增 2	$N = \text{minor}/2$
<i>mtdrN</i>	1 到 33，每次递增 2	$N = (\text{minor} - 1)/2$
<i>mtdblockN</i>	0 到 16，每次递增 1	$N = \text{minor}$

注 1: 参见第三章“MTD”中对 MTD 用户模块的定义。

表 7-2: MTD/dev 条目、次要编号，以及命名方式（续）

/dev 条目	次要编号的范围	命名方式
<i>nftlLN</i>	0 到 255、每 16 个数字为一组	$L = \text{set};^a N = \text{minor} - (\text{set} - 1) * 16$; 如果 N 的值为零，则不会附加到条目名称
<i>ftlLN</i>	0 到 255、每 16 个数字为一组	如同 <i>NFTL</i>

a. 如同 */dev* 目录中其他可分区的块设备条目，会以字母来区分不同组的设备。“a”是指第一组设备，“b”是指第二组设备，“c”是指第三组设备，依此类推。

以下是可以使用的各种 MTD */dev* 条目：

mtdN
每个条目就是一个独立的 MTD 设备或分区。记住，每个 MTD 分区如同一个独立的 MTD 设备。

mtdrN
每个条目就是相应 */dev/mtdN* 条目的只读版本。

mtdblockN
每个条目就是相应 */dev/mtdN* 项目的块设备版本。

nftlLN
每组（设备）就是一个独立的 NFTL 设备，在同组（设备）中的每个条目就是（NFTL）设备上的一个分区。在同组（设备）中的第一个条目就是一整个（NFTL）设备。例如，*/dev/nftlb* 代表第二个 NFTL 设备的全部，*/dev/nftlb3* 代表二个 NFTL 设备的第三个分区。

ftlLN
如同 NFTL。

正如我们稍后所见，并不需要在主机上手动建立这些条目。然而，将需要在目标板的根文件系统上建立若干条目，以便使用相应的 MTD 用户模块。同时请注意，前面描述的命名方式与前面提到的 *devices.txt* 文件的说明可能会有一些差异。别忘了，在此处看到的命名方式，仅是它的一种实例。

设定内核配置

正如第五章所说，MTD 子系统的配置是内核配置选项主菜单的一部分。不管你是使用基于 curses 的终端配置菜单或是通过基于 Tk 的 XWindow 配置菜单来设定内核的配置，都得进入 Memory Technology Devices（MTD）次级菜单来为内核设定 MTD 子系统的配置。

MTD次级菜单包含了一串配置选项，可以选择：将它建立成内核的一部分、将它建立成独立的模块，或根本不使用。下面是 MTD 次级菜单中可以设定的主要选项：

Memory Technology Device (MTD) support, CONFIG_MTD

要支持 core MTD 子系统必须启用此选项。如果停用此选项，则此内核不会提供任何 MTD 支持。当此选项被设为建立成模块，将会产生一个名为 *mtdcore.o* 模块。

MTD partitioning support, CONFIG_MTD_PARTITIONS

要让 MTD 设备能够划分成各个独立的分区必须启用此选项。如果将此选项设为建立成模块，则会产生一个名为 *mtddpart.o* 的模块。请注意，MTD 的分区功能对 DOC 设备不适用。要对这些设备进行分区，可以使用一般的磁盘分区设备。

Direct char device access to MTD devices, CONFIG_MTD_CHAR

这是字符设备 MTD 用户模块（就是你看到的 */dev/mtdN* 和 */dev/mtdrN*）的配置选项。如果将此选项设为建立成模块，则会产生一个名为 *mtdchar.o* 的模块。

Caching block device access to MTD devices, CONFIG_MTD_BLOCK

这是读写块设备 MTD 用户模块（就是你看到的 */dev/mtdblockN*）的配置选项。如果将此选项设为建立成模块，则会产生一个名为 *mtdblock.o* 的模块。

Read-only block device access to MTD devices, CONFIG_MTD_BLOCK_RO

这是只读块设备 MTD 用户模块（就是你看到的、与读写块设备 MTD 同名的 */dev* 条目）的配置选项。如果将此选项设为建立成模块，则会产生一个名为 *mtdblock_ro.o* 的模块。

FTL (Flash Translation Layer) support, CONFIG_FTL

要让内核支持 FTL 用户模块必须设定此选项。如果将此选项设为建立成模块，则会产生一个名为 *ftl.o* 模块。FTL 用户模块可通过 */dev/ftlN* 设备条目来存取。

NFTL (NAND Flash Translation Layer) support, CONFIG_NFTL

要让内核支持 NFTL 用户模块必须设定此选项。如果将此选项设为建立成模块，则会产生一个名为 *nftl.o* 的模块。NFTL 用户模块可通过 */dev/nftlN* 设备条目来存取。

Write support for NFTL (BETA), CONFIG_NFTL_RW

想要写入具有 NFTL 格式的设备必须启用此选项。此选项只会影响 NFTL 用户模块的建立方式，它本身并不是一个独立的用户模块。

警告： 请注意，前面提到的两种块设备 MTD 用户模块，只能择其中一个启用，尽管你同时设定了这两个选项（*mtdblock.o* 和 *mtdblock_ro.o*）。换言之，如果将读写块设备用户模块建立到内核里（不是建立成模块），你将无法启用只读块设备用户模块，不管是建立到内核里或是建立成模块。正如我们前面所见，这两种块设备 MTD 用户模块会使用相同的 */dev* 条目，因此无法同时启用。

前面的选项清单基本上都是由我前面提到的用户模块组成的。其余的 MTD 用户模块、JFFS 和 JFFS2，并不会被归类为 MTD 子系统的配置。你可以通过 File systems 次级菜单来设定它们。尽管如此，要启用 JFFS 或 JFFS2 的支持必须先启用 MTD 的支持。

MTD 次级菜单还包含四个次级菜单，可用来设定支持实际 MTD 硬件的设备驱动程序：

RAM/ROM/Flash chip drivers

包含以下配置选项：CFI-Compliant flash、JEDEC-compliant flash、old non-CFI flash、RAM、ROM 及 absent 等芯片。

Mapping drivers for chip access

mapping 驱动程序的配置选项。你可以看到一个通用的 mapping 驱动程序，要设定它必须提供设备的物理起始地址、空间大小（以十六进制表示）以及总线宽度（以字节为单位）等参数。你还可以看到目前支持 mapping 驱动程序的板子。

Self-contained MTD device drivers

包含以下配置选项：uncached system RAM、virtual memory test driver、block device emulation driver 及 DOC 等设备。

NAND Flash Device Drivers

包含 non-DOC NAND flash 设备的配置选项。

在你配置内核的 MTD 子系统之前，确定你已经看过第三章对 MTD 子系统的说明，因为此处提到的选项，前面已经在该处详细探讨过了。

当你为主机配置内核时，把所有 MTD 子系统选项都设成模块比较有利，因为这样你可以测试不同的设备设置。然而对目标板来说，用来支持固态存储设备的所有选项都必须建立成内核的一部分，而不是建立成模块。否则目标板将无法从固态存储设备安装它的根文件系统。如果忘了将目标板的内核设成可以从 MTD 设备安装它的根文件系统，内核将在启动期间发生内核恐慌情况，并且以如下的信息抱怨它无法安装根文件系统：

```
Kernel panic: VFS: unable to mount root fs on ...
```

MTD 工具程序

因为 MTD 子系统的功能不同于其他的内核子系统，所以需要一组能与之交互的特殊工具。稍后我们将会看到如何取得及安装这些工具。现在让我们来看看有哪些工具可用以及它们的用途。

警告： MTD 工具程序是一套非常有用的工具。使用这些工具之前，请先了解它们的操作原理。同时确定你了解这些工具操作的设备的特性。例如，必须小心操作 DOC 设备。如果 MTD 工具使用不当，很容易损坏 DOC 设备。

MTD 工具集里的工具可以根据其操作的 MTD 子系统组件加以分类：

一般工具

这类工具可用来操作各种类型的 MTD 设备：

info device

列出 device 设备与抹除范围有关的信息。

erase device start_address number_of_blocks

从指定的地址开始抹除 device 设备中一定数量的块。

eraseall [options] device

抹除整个 device 设备。

unlock device

为 device 设备的所有扇区解锁（注 2）。

lock device offset number_of_blocks

锁定 device 设备上一定数量的块。

fcv [options] filename flash_device

将文件复制到 flash_device 闪存设备。

doc_loadbios device firmware_file

将引导加载程序写入 device 设备的引导存储区间。尽管此命令通常只会用在 DOC 设备上，但是它并非 DOC 的专用命令。

mtd_debug operation [operation_parameters]

提供有用的 MTD 调试操作。

文件系统建立工具

这类工具将会建立文件系统供稍后相应的 MTD 用户模块使用：

mkfs.jffs2 [options] -r directory -o output_file

从 directory 目录开始建立 JFFS2 文件系统的映像文件。

注 2： 有些设备为了避免意外写入，会使用写入锁定功能。这些设备或它的某部分一旦被锁定，除非解锁否则无法被写入。

```
mkfs.jffs [options] -d directory -o output_file
```

从 directory 目录开始建立 JFFS 文件系统的映像文件。

```
jffs2reader image [options] path
```

列出 JFFS2 文件系统的映像文件中 path 路径的内容。

NFTL 工具

这类工具可与 NFTL 分区交互：

```
nftl_format device [start_address [size]]
```

将 device 设备格式化供 NFTL 用户模块使用。

```
nftldump device [output_file]
```

将 NFTL 分区的内容转储到 output_file 文件中。

FTL 工具

这类工具可与 FTL 分区交互：

```
ftl_format [options] device
```

将 FTL 设备格式化。

```
ftl_check [options] device
```

检查并提供与 FTL 设备有关的信息。

NAND 芯片工具

这类工具可用来操作 NAND 芯片：

```
nandwrite device input_file start_address
```

将 input_file 文件的内容写入 NAND 芯片。

```
nandtest device
```

测试 NAND 芯片，包括那些在 DOC 设备里的 NAND 芯片。

```
nanddump device output_file offset number_of_bytes
```

将 NAND 芯片的内容转储到 output_file 文件中。

这些工具多半会用来操作 */dev/mtdN* 设备（也就是与各类 MTD 设备交互的字符设备）。我将会描述对接下来几章极重要的 MTD 工具的典型用法。本书的编排方式如下：本章说明实际的 MTD 硬件，第八章说明根文件系统的设置，第九章说明引导加载程序的设置。

为主机安装 MTD 工具程序

MTD 工具程序是 MTD 开发 CVS 的一部分。你可以使用 CVS 取回最新版的工具。你也

可以使用 `ftp://ftp.uk.linux.org/pub/people/dwmw2/mtd/cvs/` 下载 CVS 快照（当日最新版）。以我的 DAQ 模块为例，我使用的是日期标注为 2002-07-31 的快照。

先将选择的快照下载至 `${PRJROOT}/build-tools` 并将它解开。然后移往工具程序的源码目录，开始编译这些工具：

```
$ cd ${PRJROOT}/build-tools/mtd/util
$ automake --foreign; autoconf
$ ./configure --with-kernel=/usr/src/linux
```

执行 `configure` 命令的时候，因为编译这些工具需要用到内核，所以必须指定内核源码树的路径。因为你是为主机建立这些工具，所以你提供的必须是主机内核源码树的路径。缺省情况下，内核源码树位于 `/usr/src/linux`。

正如其他的建立工作，`configure` 可用来建立编译这些工具需要的 Makefile。接着你可以编译这些工具：

```
$ make clean
$ make
```

确定你执行了 `make clean` 命令，因为必须删除工具程序源码树中若干文件，才能建立同名的符号链接，让它指向内核源码树中相应的文件。

建立的过程中，如果 `compr.c` 编译失败是因为 `KERN_WARNING` 符号没有定义，将需要编辑 `Makefile.am` 文件，并且将：

```
compr.o: compr.c
    $(COMPILE) $(CFLAGS) $(INCLUDES) -Dprintk=printf \
    -DKERN_NOTICE= -c -o $@ $<
```

替换成：

```
compr.o: compr.c
    $(COMPILE) $(CFLAGS) $(INCLUDES) -Dprintk=printf \
    -DKERN_WARNING= -DKERN_NOTICE= -c -o $@ $<
```

一旦完成变更之后，必须在执行 `make distclean` 命令之后，重新开始建立的工作，方法前面已经说过了。

工具程序建立好之后，接着把它们安装到你摆放工具的目录：

```
$ make prefix=${PREFIX} install
```

这会把工具程序安装到 `${PREFIX}/sbin` 目录。如果 `${PREFIX}/sbin` 目录并非搜索路径的一部分，还需要将此目录加入其中。参考第四章在探讨如何安装 uClibc 的工具程序时对新增目录到搜寻路径所做的完整说明。

工具程序安装好之后，接着需要在主机的 */dev* 目录中建立 MTD 设备条目。你可以在 *util* 目录中找到用来建立设备文件的 *MAKEDEV* 命令脚本。如果对设备文件的建立有兴趣，可以检查这个命令脚本的内容。*MAKEDEV* 主要会建立将在第七章提到的 */dev* 条目。因为 *MAKEDEV* 会用到 *mknod* 命令，所以你需要以 *root* 的身份来执行它：

```
# ./MAKEDEV
```

尽管你日后有可能需要更新 MTD 工具集，但是你应该不需要再次使用 *MAKEDEV*。如果可以在主机上存取 MTD 设备，因为你使用的是我在第二章提到的“可抽换存储设备设置”或“独立式设置”，可以立即操作这些 MTD 设备。如果使用的是“连接式设置”或想在“可抽换存储设备设置”中使用目标板上的 MTD 工具程序，下一节将会告诉你为目标板建立 MTD 工具程序的方法。

为目标板安装 MTD 工具程序

要为目标板安装 MTD 工具程序，首先需要下载并安装 *zlib*。前面当你为主机安装 MTD 工具程序的时候，并不需要安装 *zlib*，因为 *zlib* 是主流发行套件的一部分。*zlib* 可以从 <http://www.gzip.org/zlib/> 取得。以我的 DAQ 模块为例，我使用的是 *zlib* 1.1.4。

下载 *zlib* 并将它解开放到 *\$(PRJROOT)/build-tools* 目录。然后移往该链接库的源码目录，准备进行编译的工作：

```
$ cd $(PRJROOT)/build-tools/zlib-1.1.4
$ CC=i386-linux-gcc LDSHARED="i386-linux-ld -shared" \
> ./configure --shared
```

缺省情况下，*zlib* 会建立静态的链接库。要将 *zlib* 建立成共享链接库，当你在调用 *configure* 的时候必须设定 *LDSHARED* 变量以及提供 *--shared* 选项。建立好 *Makefile* 之后，接着编译和安装链接库：

```
$ make
$ make prefix=$(TARGET_PREFIX) install
```

如同前面安装的其他目标板链接库，我们会将 *zlib* 安装到 *\$(TARGET_PREFIX)/lib*。链接库安装好之后，接着将它安装到目标板的根文件系统：

```
$ cd $(TARGET_PREFIX)/lib
$ cp -d libz.so* $(PRJROOT)/rootfs/lib
```

建立好 MTD 工具程序之后，接着将 MTD snapshot 下载并解开放到 *\$(PRJROOT)/sysapps*。然后移往工具程序的源码目录开始建立工具：

```
$ cd $(PRJROOT)/sysapps/mtd/util
$ automake --foreign; autoconf
```

```
$ CC=i386-linux-gcc ./configure \  
> --with-kernel=${PRJROOT}/kernel/linux-2.4.18  
$ make clean  
$ make
```

以此例来说，内核的路径指向目标板使用的内核。正如前一节所说，建立的过程中，如果 *compr.c* 编译失败是因为 `KERN_WARNING` 符号没有定义，将需要编辑 *Makefile.am* 文件以便修正此问题，并且在下达 *make distclean* 这条命令之后才重新开始建立的工作。

工具程序建立好之后，接着将它们安装到目标板的根文件系统：

```
$ make prefix=${PRJROOT}/rootfs install
```

这将会把工具程序安装到 *\${PRJROOT}/rootfs/sbin*。然而此处将不会执行 *MAKEDEV* 命令脚本，因为它不适合用来在根文件系统以外的地方建立设备条目。如果想以这个命令脚本来为目标板建立设备条目，必须把它复制到目标板的根文件系统，然后在目标板运行的时候将它执行一次。然而，这将会浪费目标板上的文件系统资源，因为这个命令脚本会为你在系统中可能用到的所有 MTD 设备建立 */dev* 条目。我们将会在接下来的各节中看到如何在目标板上建立需要的设备条目。

NOR 和 NAND flash 的工作原理

flash 设备，包括 NOR flash 设备（例如 CFI flash 芯片）以及 NAND flash 设备（例如 DOC），与磁盘存储设备不同。你无法任意读写这些设备。想知道如何正确操作 flash 芯片，我们首先必须了解它们的工作原理。flash 设备通常会被划分成若干“抹除块”。最初会有一个所有位全被设为 1 的空白块。对此块进行写入操作等于将位清为 0（译注 1）。一旦块中所有位全被清除（设为 0），抹除此块的惟一可能方式，就是将它的所有位同时设为 1。以 NOR flash 设备来说，“抹除块”中的每个位可以被个别设为 0，直到整个块的所有位全被设为 0。相对来说，NAND flash 设备会将“抹除块”进一步切割成页，它的典型长度为 512 字节。在页被写入特定次数（通常少于 10 次）之后，它们的内容将会变得无法确定。只有在页隶属的块被完全抹除之后，才能再次使用相应的页。

原生的 CFI（Common Flash Interface）flash 设备

最近不管是小型或中型的非 X86 嵌入式 Linux 系统都配备某种形式的 CFI flash 设备。设置 Linux 使用的 CFI flash 设备相当简单。这一节我们将会探讨如何在 Linux 中设置以及

译注 1：若数据中相应的位为 1，则不会进行写入操作。

操作 CFI 设备。然而，我们将不会讨论如何在此类设备上使用文件系统，因为这是下一章要探讨的主题。接下来，我将尽可能依照在 Linux 中使用 CFI flash 设备涉及的实际步骤来分节说明。尽管如此，仍然可以根据目前的操作选择自己需要的提示。

配置内核

要使用 CFI flash 设备，将需要启用内核对下列选项的支持：

- Memory Technology Device (MTD) support 选项
- MTD partitioning support 选项，如果想对 flash 设备进行分区
- Direct char device access to MTD devices 选项
- Caching block device access to MTD devices 选项
- RAM/ROM/Flash chip drivers 次级菜单里的 Detect flash chips by Common Flash Interface (CFI) probe 选项
- Mapping drivers for chip access 次级菜单里特定卡（板）的 CFI flash device mapping driver 选项

当然你还可以启用其他选项，不过以上所列的是最基本的条目。此外，如果想让内核从 CFI 设备安装它的根文件系统，切记要将这些选项设成“y”而不要设成“m”。

进行分区

与磁盘或 DOC 设备不同的是，一般不能使用 *fdisk* 或 *pdisk* 之类的工具对 CFI flash 设备进行分区，因为分区信息通常无法存储在 CFI flash 设备上。取而代之的是，设备的分区信息会硬编码在 mapping 驱动程序里，并且会在设备初始化期间向 MTD 子系统注册。实际的设备根本不会包含任何的分区信息。因此必须编辑 mapping 驱动程序的 C 程序源码才能修改其分区。

例如，我的控制模块适合使用 TQM8xxL PPC 板。这类板子最多可以包含两组 4 MB 的 flash 存储库。每组宽度 32 位的 flash 存储库由两个宽度 16 位的 flash 芯片组成。为了定义这些板子的分区，与其相应的 mapping 驱动程序将会进行如下数据结构的初始化：

```
static struct mtd_partition tqm8xxl_partitions[] = {
    {
        name:      "ppcboot",                /* PPCBoot 固件 */
        offset: 0x00000000,
        size:      0x00040000,                /* 256 KB */
    },
    {
        name:      "kernel",                  /* 缺省内核映像 */
        offset: 0x00040000,
```

```
        size:    0x000C0000,
    },
    {
        name:     "user",
        offset:   0x00100000,
        size:     0x00100000,
    },
    {
        name:     "initrd",
        offset:   0x00200000,
        size:     0x00200000,
    }
};

static struct mtd_partition tqm8xxl_fs_partitions[] = {
    {
        name:     "cramfs",
        offset:   0x00000000,
        size:     0x00200000,
    },
    {
        name:     "jffs2",
        offset:   0x00200000,
        size:     0x00200000,
    }
};
```

此例中，tqm8xxl_partitions 为第一组 4 MB 的 flash 存储库定义了四个分区，tqm8xxl_fs_partitions 为第二组 4 MB 的 flash 存储库定义了两个分区。每个分区的定义包含三个属性：name、offset 和 size。

分区的 name 属性只是一个助记字符串。MTD 子系统及 MTD 工具程序并无法在相应的分区上使用此字符串。offset 属性可为 MTD 子系统提供分区的起始地址，size 属性的用途应该不用多做解释。请注意，设备上每个分区的起始地址就是前一个分区的结束地址；不会有任何间隙。表 7-3 展示了 TQM860L 板上物理存储地址范围、其包含的两组 4 MB 的 flash 存储库被连续配置在从 0x40000000 开始的地址上。

表 7-3: TQM860L 板上 flash 存储库的分区的物理存储配置

设备	起始地址	结束地址	分区名称
0	0x40000000	0x40040000	ppcboot
0	0x40040000	0x40100000	kernel
0	0x40100000	0x40200000	user
0	0x40200000	0x40400000	initrd
1	0x40400000	0x40600000	cramfs
1	0x40600000	0x40800000	jffs2

在此设备的 mapping 驱动程序注册期间，内核将会显示如下的信息：

```
TQM flash bank 0: Using static image partition definition
Creating 4 MTD partitions on "TQM8xxL Bank 0":
0x00000000-0x00040000 : "ppcboot"
0x00040000-0x00100000 : "kernel"
0x00100000-0x00200000 : "user"
0x00200000-0x00400000 : "initrd"
TQM flash bank 1: Using static file system partition definition
Creating 2 MTD partitions on "TQM8xxL Bank 1":
0x00000000-0x00200000 : "cramfs"
0x00200000-0x00400000 : "jffs2"
```

你可以借着检查 */proc/mtd* 来察看分区。下面是我的控制模块显示的分区信息：

```
# cat /proc/mtd
dev:      size  erasesize  name
mtd0: 00040000 00020000 "ppcboot"
mtd1: 000c0000 00020000 "kernel"
mtd2: 00100000 00020000 "user"
mtd3: 00200000 00020000 "initrd"
mtd4: 00200000 00020000 "cramfs"
mtd5: 00200000 00020000 "jffs2"
```

请注意，分区必须以抹除单位为界。因为 flash 芯片以块（而非字节）为单位，所以建立分区的时候必须考虑到“抹除块”的大小。此例中，“抹除块”的大小为 128 KB，因此所有的分区必须以 128 KB（0x20000）为界。

另一个提供 MTD 分区信息的方法

现在 MTD 子系统已经能够从 ARM 架构的内核引导选项取得分区信息。iPAQ Familiar 发行套件便是通过这种功能将 CFI flash 芯片的分区信息提供给 iPAQ 内核。

近来，此能力涵盖所有架构的通用形式已经被整合进主 MTD 源码的 CVS 库。尽管撰写本书时这些变动尚未纳入主内核源码树中，不过它们终究是会被整合进去的，因此 Linux 迟早都会让所有架构通过内核引导选项传递分区信息。

承前例，下面这些引导选项为内核提供了 TQM8xxL 板的分区信息（因为版面编排的关系，原本是一行的引导选项被分成两行，实际应用时必须指定成一行）：

```
mtdparts=0:256k(ppcboot)ro,768k(kernel),1m(user),-(initrd);1:2m(cramfs),-
(jffs2)
```

必要的 /dev 条目

要存取 CFI flash 设备，需要为字符设备和块设备“MTD 用户模块”建立 */dev* 条目。而且你必须根据设备上的分区数来为这两种 MTD 用户模块建立相同数量的 */dev* 条目。举例来说，下列命令将会为我的 TQM860L 板上六个分区建立根文件系统条目：

```
$ cd ${PRJROOT}/rootfs/dev
$ su -m
Password:
# for i in $(seq 0 5)
> do
> mknod mtd$i c 90 $(expr $i + $i)
> mknod mtdblock$i b 31 $i
> done
# exit
```

结果会产生以下项目：

```
$ ls -al mtd*
crw-rw-r-- 1 root root 90, 0 Aug 23 17:19 mtd0
crw-rw-r-- 1 root root 90, 2 Aug 23 17:20 mtd1
crw-rw-r-- 1 root root 90, 4 Aug 23 17:20 mtd2
crw-rw-r-- 1 root root 90, 6 Aug 23 17:20 mtd3
crw-rw-r-- 1 root root 90, 8 Aug 23 17:20 mtd4
crw-rw-r-- 1 root root 90, 10 Aug 23 17:20 mtd5
brw-rw-r-- 1 root root 31, 0 Aug 23 17:17 mtdblock0
brw-rw-r-- 1 root root 31, 1 Aug 23 17:17 mtdblock1
brw-rw-r-- 1 root root 31, 2 Aug 23 17:17 mtdblock2
brw-rw-r-- 1 root root 31, 3 Aug 23 17:17 mtdblock3
brw-rw-r-- 1 root root 31, 4 Aug 23 17:17 mtdblock4
brw-rw-r-- 1 root root 31, 5 Aug 23 17:17 mtdblock5
```

抹除

对 CFI flash 设备进行写入操作之前，需要先抹除其内容。方法是使用 MTD 工具程序提供的抹除命令：*erase* 和 *eraseall*。

举例来说，在更新我的控制模块上的最初 RAM disk 之前，我必须先抹除 *initrd* 分区：

```
# eraseall /dev/mtd3
Erased 2048 Kibyte @ 0 -- 100% complete.
```

写入与读出

利用将“抹除块”里的位连续设为 0 的能力，使 JFFS2 之类的 flash 文件系统得以轻易对它进行存取，但是你通常无法用用户层的工具对 MTD 设备进行写入操作。举例来说，如果想要使用 MTD 设备或分区的原始字符型 */dev* 条目来更新它的内容，在写入新数据之前，通常必须抹除此设备或分区。

要将数据写入原始的 flash 设备可以使用传统的文件系统命令，例如 *cat* 和 *dd*。举例来说，在控制模块的 *initrd* 分区被抹除之后，我会使用如下的命令将最初的 RAM disk 映像写入相应的分区：

```
# cat /tmp/initrd.bin > /dev/mtd3
```

此例中，目标板通过 NFS 安装根文件系统，而且 MTD 命令在目标板上执行。我不仅可以使使用 *cat* 命令，还可以使用 *dd* 命令，这两个命令最后都可以获得相同的结果。

读取 CFI MTD 分区跟读取任何其他设备并没有什么不同。例如在控制模块上执行下列命令，可将 *bootloader* 分区的二进制映像复制到文件中：

```
# dd if=/dev/mtd0 of=/tmp/ppcboot.img
```

因为 *bootloader* 映像本身可能不会填满整个分区，所以 *ppcboot.img* 文件中除了 *bootloader* 映像可能会包含若干无关的额外数据。

DiskOnChip

DOC 设备广泛用于以 x86 为基础的嵌入式 Linux 系统中，MTD 子系统对 DOC 设备的支持也有一段不算短的时间。例如，我会在 DAQ 模块中使用 DOC 设备。然而，DOC 犹如一只奇兽需要主人细心照料。你很快就会知道我这么说的原因了。

预备工作

与嵌入式 Linux 系统中可以找到的大多数其他设备不同，要能善用任何 DOC 设备，自己至少必须准备一张 DOS 引导磁盘，内含 M-Systems 的 DOS DOC 工具。这么做基本上有两个理由：

- 如同所有的 NAND flash 设备，DOC 设备可能会包含若干制造上的缺陷而导致坏块。每个 DOC 设备在出厂之前都会被写入一个 *Bad Block Table*（坏块表、BBT）。尽管 BBT 并没有写保护功能，不过它是所有读写 DOC 软件在操作时必备的数据。M-Systems 的 DOC 软件可以读取 BBT 并且将它存入文件中。然而，目前 Linux 并没有任何工具程序可用来取回 BBT。
- 2.4.x（至少一直到 2.4.19）版内核包含的 NFTL 驱动程序无法处理 M-Systems 某些版本的 DOC 固件。如果固件是 5.0 以及之后的版本，非常可能发生问题。因此，可能需要使用 M-Systems 的工具将 DOC 目前的固件替换成较旧的版本，让 Linux 能够正确操作 DOC 设备。目前，M-Systems 的 TrueFFS 工具提供的 4.2 版固件适合在所有内核上使用。

此外,有两种方法可用来将bootloader安装到DOC以及对它进行NFTL格式化。第一个方法(也是MTD维护者最建议采用的方法)就是使用M-Systems的DOS工具程序*dformat*。第二个方法(可让你在Linux中对DOC设备获得最大程度的控制)就是使用MTD工具程序*doc_loadbios*和*nftl_format*。接下来让我们讨论这两个方法。

M-Systems的DOS DOC工具以及相关的文件可以从该公司位于<http://www.m-sys.com/>的网站取得。如果准备使用Linux的方法将bootloader安装在DOC上以及对它进行格式化,则需要4.2版以及5.0或之后版本的M-Systems工具。如果准备使用DOS的方法,只需要5.0或之后版本的M-Systems工具。在我撰写本书时M-Systems工具最新的版本是5.1.2。下面所举的例子基于32 MB的DOC 2000设备。例子中各工具的输出结果取决于实际使用的设备,不过应该跟此处列示的输出结果非常类似。

首先准备好一张内含最新工具的DOS磁盘。DOS磁盘备妥后,接着以该磁盘开启包含DOC设备的系统。然后以如下的方式使用这些工具(注3):

1. 使用5.0或之后版本的DOC工具,产生BBT的副本:

```
A:\>dformat /win:d000 /noformat /log:docbbs.txt
DFORMAT Version 5.1.0.25 for DOS
Copyright (C) M-Systems, 1992-2002

DiskOnChip 2000 found in 0xd0000.
32M media, 16K unit

OK
```

*dformat*命令通常会被用来对DOC进行DOS格式化。此例中,我们以*/noformat*选项指示*dformat*不要进行格式化的操作。此外,我们会指示它将设备位于segment 0xD000(注4)的BBT记录到*docbbs.txt*文件中。一旦*dformat*取出BBT,请将*docbbs.txt*的副本保存在一个安全的地方,因为如果在Linux中不小心抹除了整个

注3: 这些工具之语义和用法的完整说明可参考M-Systems操作手册。

注4: PC上的实模式(real-mode,译注2)地址采segment:offset的寻址方式。每当offset的值为零的时候,这个寻址法通常可以只提供segment的值。此列中,segment 0xD000的起始地址是0xD0000,正如*dformat*的输出所示。

译注2: 实模式(real-mode)指的是早期8086/8088 CPU的寻址方式:因为只有20条地址线,所以只能寻址1MB的内存空间,又因为CPU的缓存器宽度只有16位所以采用segment:offset的寻址方式。在实模式之下,1MB的内存空间可以被分割成16个64KB(segment),所以你可以先用16位的缓存器指出想要寻址的segment,再以另一个16位的缓存器(代表offset)指出这个segment里的确切地址。


```
DFORMAT Version 3.3.9 for DiskOnChip 2000 (V4.2)
Copyright (C) M-Systems, 1992-2000

Driver not loaded - using direct access
WARNING: All data on DiskOnChip 2000(R) will be destroyed. Continue ? (Y/N)y

Medium physical size is 32768 KBytes
Boot-image size is 48 KBytes

Finished 32768 KBytes
Writing Boot-Image
Format complete. Formatted size is 32032 KBytes.
Please reboot to let DiskOnChip 2000(R) install itself.
```

一旦你以4.2版的固件完成了芯片的格式化后,建议先将系统关机然后再引导。通常一个简单的重新引导动作就可以适切地完成固件的安装,不过有些系统却需要完整的关机然后再引导。

现在你已经准备好可以在Linux上使用DOC设备。

设定内核配置

写作本书时,如果使用DOS的方法来安装bootloader以及格式化DOC设备,需要从MTD CVS库取得最新的MTD程序代码,并以此来修补内核。MTD计划位于<http://www.linux-mtd.infradead.org>的网站会告诉你如何从CVS库取回程序代码。

要使用DOC设备,需要让内核支持以下选项:

- Memory Technology Device (MTD) support 选项
- MTD partitioning support 选项、如果想要对flash设备进行分区
- Direct char device access to MTD devices 选项
- NFTL (NAND Flash Translation Layer) support 选项
- Write support for NFTL (BETA)选项
- Self-contained MTD device drivers 次级菜单里的 M-Systems Disk-On-Chip 2000 and Millennium 选项

如同CFI flash,自己也可以选择其他的选项。如果将以上列示的条目编译成模块,对DOC的支持将会分别放在 *ocecc.o*、*doc2000.o* 和 *docprobe.o* 三个文件中。此时执行 *modprobe docprobe* 这道命令,应该可以自动加载这三个模块。对DOC的支持不论是内核的一部分还是加载成模块,DOC探测驱动程序都会为DOC设备分析可能的内存地址。探测驱动程序会针对它分析的每个内存地址输出它找到的相关信息。下面是我的DAQ模块上探测驱动程序的实际输出结果:


```

Possible DiskOnChip with unknown ChipID FF found at 0xc8000
...
Possible DiskOnChip with unknown ChipID FF found at 0xce000
DiskOnChip 2000 found at address 0xD0000
Flash chip found: Manufacturer ID: 98, Chip ID: 73 (Toshiba TH58V128DC)
2 flash chips found. Total DiskOnChip size: 32 MiB
Possible DiskOnChip with unknown ChipID FF found at 0xd2000
Possible DiskOnChip with unknown ChipID FF found at 0xd4000
...

```

M-Systems 的 DOC 驱动程序

M-Systems 为 Linux 提供的 DOC 驱动程序就放在它们的 Linux 工具套件里。然而，这个驱动程序的发行并非采用 GPL 的许可方式，而且你只能将它当成可加载的内核模块来用。发布内建此驱动程序的内核事实上违背了 GPL 的原则。因此，如果想让使用了 M-Systems 驱动程序的内核从 DOC 引导，需要在最初使用 RAM disk 加载这个二进制驱动程序。此外，有人曾在 MTD 邮件论坛上提到，该驱动程序会占用许多系统资源，而且有时会造成串行端口上数据的丢失。因此，建议各位不要使用 M-Systems 的 Linux DOC 驱动程序。可以改在此处提到的 GPL 版本的 MTD 驱动程序。

必要的 /dev 条目

要存取 DOC 设备，必须为字符设备和 NFTL “MTD 用户模块” 建立 /dev 条目。你必须为系统上的 DOC 设备建立相应的字符设备条目以及 NFTL 条目。对每一组 NFTL 来说，必须根据设备上的分区数目建立相同数量的条目。以我的 DAQ 模块为例，因为我使用了一个 DOC 设备，而且它只有一个分区，所以我会使用下列命令来建立相关条目：

```

$ cd ${PRJROOT}/rootfs/dev
$ su -m
Password:
# mknod mtd0 c 90 0
# mknod nftla b 93 0
# mknod nftla1 b 93 1
# exit

```

这会产生如下的条目：

```

$ ls -al mtd* nftl*
crw-rw-r-- 1 root root 90, 0 Aug 29 12:48 mtd0
brw-rw-r-- 1 root root 93, 0 Aug 29 12:48 nftla
brw-rw-r-- 1 root root 93, 1 Aug 29 12:48 nftla1

```

抹除

抹除 DOC 设备的方式跟其他的 MTD 设备一样，也是使用 *erase* 和 *eraseall* 命令。在你对 DOC 设备执行这类命令之前，确定你有为 BBT 留下副本，因为 DOC 设备遭抹除时，将会清除其中包含的 BBT。

例如，为了将 DAQ 模块中的 DOC 设备整个抹除，我会在 DAQ 模块上使用如下的命令：

```
# eraseall /dev/mtd0
Erased 32768 Kibyte @ 0 -- 100% complete.
```

通常，只有在你想要抹除 bootloader 以及设备上的当前格式时才需要抹除 DOC 设备。举例来说，如果安装了一个 Linux bootloader，但事后回过头想使用 M-Systems 的 SPL，于是在你用 M-Systems 的工具安装 M-Systems 的 SPL 之前，将需要使用 *eraseall* 命令。然而，在抹除整个设备之后，需要用 M-Systems 的工具恢复 BBT。

安装 bootloader 映像

如果目标板无法从 DOC 设备引导，可以跳过这个步骤。否则你需要在进行下一个步骤之前建立 bootloader（参见第九章的说明）。尽管如此，首先让我们检查系统如何从 DOC 引导。

X86 系统启动期间，BIOS 会在内存中寻找 BIOS extensions，只要找得到 BIOS 就会加以执行。DOC 设备中包含了一个称为 *Initial Program Loader* (IPL) 的 ROM 程序，IPL 便是利用此特性去安装另一个称为 *Secondary Program Loader* (SPL) 的程序，SPL 可作为系统启动期间的 bootloader。缺省情况下，SPL 由 M-Systems 自己的固件提供。然而，要让 Linux 从 DOC 设备引导，SPL 必须被换成能够认得 DOC 设备上 Linux 格式的 bootloader。我们将会在第九章讨论各种具有 DOC 能力的 Linux bootloader。现在让我们来看一下如何在 DOC 上安装自己的 SPL。

下面是 Linux 中我用来在 DOC 上安装 GRUB bootloader 映像的命令，*grub_firmware*：

```
# doc_loadbios /dev/mtd0 grub_firmware
Performing Flash Erase of length 16384 at offset 0
Performing Flash Erase of length 16384 at offset 16384
Performing Flash Erase of length 16384 at offset 32768
Performing Flash Erase of length 16384 at offset 49152
Performing Flash Erase of length 16384 at offset 65536
Performing Flash Erase of length 16384 at offset 81920
Writing the firmware of length 92752 at 0... Done.
```

下面是 DOS 中我用来在 DOC 上安装 GRUB bootloader 映像的命令：

```
A:\>dformat /win:d000 /bdkf0:grub_firmware
DFORMAT Version 5.1.0.25 for DOS
Copyright (C) M-Systems, 1992-2002
WARNING: All data on DiskOnChip will be destroyed. Continue ? (Y/N)y

DiskOnChip 2000 found in 0xd0000.
32M media, 16K unit

Formatting                2042
Writing file to BDK 0     92752
OK
Please reboot to let DiskOnChip install itself.
```

如同固件版本的更新，系统需要在使用 *doc_loadbios* 或 *dformat* 之后重新引导，以便完成固件的安装。请注意，在你阅读过第九章关于如何使用 bootloader 的说明之前，不要使用 *doc_loadbios* 或 *dformat*。

NFTL 的格式化

目前，在 Linux 中使用 DOC 设备的惟一方法就是对它进行 NFTL 格式化。一旦 DOC 设备完成了 NFTL 格式化，我们就可以使用传统的块设备工具以及与设备有关的文件系统。

如果想要从 DOC 引导，在你对 DOC 进行任何进一步的操作之前，请参考第九章中与 X86 bootloader 有关的部分。

稍早在 DOC 上安装 GRUB 的时候，如果使用的是 DOS 中的 *dformat* 工具程序，那么 DOC 已经完成了 NFTL 格式化。如果使用的是 Linux 中的 *doc_loadbios*，还必须使用 *nftl_format* 命令进行 NFTL 格式化。

下面这个 MTD 命令可以对整个 DOC 设备进行 NFTL 格式化：

```
# nftl_format /dev/mtdd0
$Id: ch07,v 1.10 2003/04/01 17:10:50 free2 Exp ldolby $
Phase 1. Checking and erasing Erase Zones from 0x00000000 to 0x02000000
        Checking Zone #2047 @ 0x1ffc000
Phase 2.a Writing NFTL Media Header and Bad Unit Table
Phase 2.b Writing Spare NFTL Media Header and Spare Bad Unit Table
Phase 3. Writing Unit Control Information to each Erase Unit
```

这个命令会花时间处理 DOC 上各个“抹除块”。一旦 *nftl_format* 在 DOC 中遇到坏块，它就会输出如下的信息：

```
Skipping bad zone (factory marked) #BLOCK_NUM @ 0xADDRESS
```

nftl_format 输出的 *BLOCK_NUM* 和 *ADDR* 应该要与前面在 *docbbt.txt* 文件中产生的内容相符。

警告： *nftl_format* 命令要能正常工作，它必须“完全控制”并且“排他存取”欲进行格式化的原始DOC设备。事实上，前面列出的命令用到了 */dev/mtdX* 设备条目，保证可以达到“完全控制”的要求。因为这些条目是由字符设备MTD用户模块负责处理，所以这些设备上进行的操作与实际的硬件之间并不存在转换层。因此，*nftl_format* 进行的任何操作都会直接影响硬件。

然而，原始DOC设备的“排他存取”有些麻烦，这与NFTL驱动程序有关。基本上，一旦NFTL驱动程序认出DOC设备的时候，它会认为自己能够完全控制该设备。因此，当一个DOC设备受到NFTL驱动程序的控制时，就不应该使用其他软件（包括*nftl_format*）来操作该设备。有若干方法可用来避免此类的冲突，这取决于你使用的内核配置。

如果NFTL驱动程序被设成模块，则执行*nftl_format*之前必须先卸该模块。一旦*nftl_format*完成设备的格式化，可以重新加载该模块。如果NFTL驱动程序被设成内建在内核中，不是使用另一个未内建NFTL驱动程序的内核，就是另外建立一个未内建NFTL驱动程序的内核。如果想要继续使用内建NFTL驱动程序的内核，可以使用*eraseall*命令抹除整个设备。抹除之后重新启动系统，内建的NFTL驱动程序便认不出DOC来了，因此不会对*nftl_format*的操作造成干扰。最后，如果首次执行这些命令，NFTL驱动程序在这个阶段应该认不出DOC设备上的任何格式，因此不会产生问题。

如果已经使用*doc_loadbios*将Linux bootloader安装到DOC，则必须跳过写入bootloader的区域，并且从它的结尾开始格式化的工作。要完成此事，必须为*nftl_format*提供一个偏移值。举例来说，在我把GRUB安装成SPL之后，我会使用下面这道命令对DOC进行NFTL格式化：

```
# nftl_format /dev/mtd0 98304
$Id: ch07.v 1.10 2003/04/01 17:10:50 free2 Exp 1dolby $
Phase 1. Checking and erasing Erase Zones from 0x00018000 to 0x02000030
        Checking Zone #2047 @ 0x1ffc000
Phase 2.a Writing NFTL Media Header and Bad Unit Table
Phase 2.b Writing Spare NFTL Media Header and Spare Bad Unit Table
Phase 3. Writing Unit Control Information to each Erase Unit
```

98304的偏移值取决于*doc_loadbios*命令的输出，参见前面的范例，此命令输出的最后一道抹除信息报告说，它从地址81920开始抹除了16384字节。因此98304是bootloader区域之后第一个可用地址。

在DOC设备完成NFTL的格式化之后，如“预备工作”中所述，必须将系统重新引导。如果NFTL驱动程序在内核启动时或加载*nftl.o*模块的时候开启，它应该会输出如下的信息：

```
NFTL driver: nftlcore.c $Revision: 1.10 $, nftlmount.c $Revision: ...
Cannot calculate an NFTL geometry to match size of 0xfea0.
Using C:J018 H:16 S:4 (== 0xfe80 sects)
```

如果 NFTL 驱动程序认出 DOC 设备来但无法辨识其格式，则会输出如下的信息：

```
Could not find valid boot record
Could not mount NFTL device
```

如果尚未对 DOC 设备使用 *nftl_format* 命令，这属于正常的信息；如果已经对 DOC 使用过 *nftl_format* 命令，这便是一个错误的信息。

如果是错误信息，后面可能会跟着如下信息：

```
Sorry, we don't support UnitSizeFactor 0x06
```

或是：

```
Sorry, we don't support UnitSizeFactor of != 1 yet.
```

会看到这些信息有很多原因。如果按照以上的指示以及第九章的指示来做，这些原因与你无关。第一种原因可能是你使用了不合适的 MTD 和 GRUB 版本。例如，当我在 DAQ 模块的 DOC 中使用了 GRUB 于 November 2002 的 CVS 版本就会看到这些信息。由于某个原因，由这个 GRUB 版本产生的固件映像会让 2.4.18 版内核的 NFTL 驱动程序感到困惑。为了解决这个问题，让 NFTL 驱动程序认出这个经 NFTL 格式化的设备，我改用 GRUB 0.92 而不使用最新的 CVS 版本。如果使用 *dformat* 将 GRUB 安装到 DOC 之上，却没有使用从 CVS 取得的最新 MTD 程序代码来修补内核，可能也会看到类似的错误信息。

每当你看到这类信息时，请确定你是遵照以上的步骤进行操作的。如果这不是操作上的错误，可以选择埋头苦干发挥 hacking 能力自己解决问题。然而，通常比较好的做法是搜索以往的 MTD 邮件论坛记录，或是向 MTD 邮件论坛提出问题，因为其他人可能也会遇到同样的问题，并且为问题找到了解决方案。当你向 MTD 邮件论坛或任何其他邮件论坛提出问题时，对问题的描述应该尽可能详尽。尤其是，应该提供相关软件组件的版本编号、指出实际的操作程序以及提供所用工具的输出结果。

建立分区

在 DOC 设备上完成 NFTL 的格式化之后，可以使用 *fdisk* 为设备建立分区。以下示范用 *fdisk* 在 NFTL 设备上建立分区的操作步骤：

```
# fdisk /dev/nftla
Device contains neither a valid DOS partition table, nor Sun or S...
Building a new DOS disklabel. Changes will remain in memory only,
until you decide to write them. After that, of course, the previous
content won't be recoverable.
```

```

Command (m for help): p

Disk /dev/nftla: 16 heads, 4 sectors, 1018 cylinders
Units = cylinders of 64 * 512 bytes

    Device Boot      Start         End      Blocks   Id  System

Command (m for help): d
Partition number (1-4): 1

Command (m for help): n
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-1018, default 1):
Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-1018, default 1018):
Using default value 1018

Command (m for help): p

Disk /dev/nftla: 16 heads, 4 sectors, 1018 cylinders
Units = cylinders of 64 * 512 bytes

    Device Boot      Start         End      Blocks   Id  System
/dev/nftla1          1         1018       32574    83  Linux

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.

WARNING: If you have created or modified any DOS 6.x
partitions, please see the fdisk manual page for additional
information.
Syncing disks.

```

请注意，建立第一个分区之前，需要先删除第一个分区。这是因为使用 *dformat* 安装 bootloader 以及对 DOC 进行格式化，也会建立一个扩及整个设备的 FAT 分区。如果之前使用的是 Linux 的 *doc_loadbios*，当你删除第一个分区时，*fdisk* 将会显示如下的错误信息，可以忽略不管：

```
Warning: partition 1 has empty type
```

同时请注意，如果不想在 DOC 或任何其他存储设备上使用单一分区，可以删除所有的分区，而且以整个设备来存储文件系统。

《Running Linux》第三章对 *fdisk* 的使用有完整的说明。为 DOC 建好分区之后，可以将 DOC 的分区当成是任何传统的磁盘分区来操作。此外，可以格式化并且安装 NFTL 分区。我们将会第八章讨论相关细节。

磁盘设备

在嵌入式 Linux 设备中操作磁盘设备（注 5）的方式如同在 Linux 工作站或服务器中一样。接下来，我们的讨论将只会专注在与传统磁盘操作不同的方面。因此，建议参考探讨一般 Linux 系统管理的其他文档或书籍，例如《Running Linux》，以便了解其余的细节。

CompactFlash

在 Linux 中，CompactFlash（CF）卡有两种存取方式：不是当成 IDE 磁盘（以 CF-to-IDE 或 CF-to-PCMCIA 适配卡的方式插入）就是当成 SCSI 磁盘（通过 USB CF 读卡器存取）。通常主机会使用 USB 读卡器来存取 CF 卡，而目标板会使用 CF-to-IDE 或 CF-to-PCMCIA 适配卡来存取该设备。因此，CF 卡在主机会被视为 SCSI 磁盘，在目标板上会被视为 IDE 磁盘。然而，经由两种不同的内核磁盘子系统存取同一张 CF 卡有可能会造成问题，正如我们在第九章为 CF 卡设定 LILO 配置期间看到的问题。当然，如果 CF 设备的存取总是经由相同的磁盘子系统则不会有问题。

主机若要经由 USB CF 读卡器存取 CF 卡，其内核必须支持 USB 存储设备。大部分发行套件会将 USB 设备的支持建立成模块。因此，必须在主机上加载适当的 USB 模块以及 SCSI 磁盘驱动程序：

```
# modprobe usb-storage
# modprobe uhci
# modprobe sd_mod
```

尽管此例中用到 *uhci* 模块，但是有些系统（例如 Apple 的系统）却需要用到 *usb-ohci*。一旦加载这些模块之后，可以在 */proc* 里的适当条目中看到 CF 读卡器。例如，我的 PC 主机上附接了一台 SanDisk SDDR-31 读卡器，以下是我通过 SCSI 子系统看到的信息：

```
# cat /proc/scsi/scsi
Attached devices:
Host: scsi0 Channel: 00 Id: 00 Lun: 00
  Vendor: SanDisk  Model: ImageMate II      Rev: 1.30
  Type:   Direct-Access                    ANSI SCSI revision: 02
# cat /proc/scsi/usb-storage-0/0
  Host scsi0: usb-storage
    Vendor: SanDisk Corporation
    Product: ImageMate CompactFlash USB
    Serial Number: None
```

注 5： 凡被 Linux 内核当成“磁盘”来用的设备，我在此处都会称之为“磁盘设备”。因此，CompactFlash 设备可被视为 ATA（IDE）磁盘。

```
Protocol: Transparent SCSI
Transport: Bulk
GUID: 078100020000000000000000
Attached: Yes
```

此例中，因为读卡器是 SCSI 总线上的第一个设备，所以可以当成 `/dev/sda` 来存取。因此，我可以把 CF 卡当成是传统的 SCSI 磁盘来进行分区、格式化以及安装：

```
# fdisk /dev/sda
...
# mkdir /mnt/cf
# mke2fs /dev/sda1
# mount -t ext2 /dev/sda1 /mnt/cf
```

CF卡需要建立多少分区以及这些分区的使用方式与目标板有很大的关系。如果目标板是 X86 PC 的衍生品，可以使用单个分区。如果目标板是使用 U-Boot bootloader 的 PPC，就需要使用若干小分区来存放内核映像，以及一个大分区来存放根文件系统。这是因为 U-Boot 可以读取 CF 设备上的分区以及这些分区上的数据，但是它认不出任何文件系统结构。因此内核映像必须写到 U-Boot 可以读取的原始分区。第九章将会讨论到以 CF 卡当成引导设备的例子。

软盘

如果想把软盘作为嵌入式 Linux 项目的主要存储设备，可参考 Tom Fawcett 撰写的《Linux Bootdisk HOWTO》(LDP) 的第六章“Putting them together: Making the diskette(s)”。Tom 详细说明了如何使用 LILO 或内核本身建立可引导的软盘。尽管你不需要阅读该 HOWTO 文件其他的章节，不过它的前提是你已经建立了一个内含根文件系统的 RAM disk。本书的第八章会说明建立这个 RAM disk 映像的方法。

我们将不会进一步讨论如何在嵌入式 Linux 系统中使用软盘，因为产品级的系统很少会用到软盘，而且《Linux Bootdisk HOWTO》已经说明得相当清楚了。

硬盘

当配置好硬盘让它能够在嵌入式 Linux 系统中使用时，目标板的引导设置最方便的方式，就是将目标板的磁盘附接至主机自己的磁盘接口。这样你就可以在主机上直接操作目标板的硬盘。

举例来说，如果主机已经有一个 IDE 磁盘被视为 `hda`，目标板的 IDE 磁盘则可能会被视为 `hdb` 或 `hdc`，这取决于主机的设置。正如任何其他硬盘一般，我们可以格式化并安装目标板的磁盘。惟一的差别是，目标板的磁盘在主机上会被视为第二个磁盘，例如 `hdb`

或 hdc，然而它在目标板上很可能会被视为 hda。当要设定 bootloader 配置的时候，这会导致一些问题。我们将会第九章进一步讨论这方面的问题。

是否启用交换功能

交换功能是大多数 Linux 工作站和服务器的基本组件。它借着在存储设备上仿真出额外的内存空间，让系统能够寻址比实际还多的内存空间。然而大多数的嵌入式存储设备，例如 flash 和 DOC 设备，并不适合这种应用，因为它们的抹除和写入周期有限。由于应用程序无法控制内核是否进行交换，所以这可能会加速存储设备的损耗。因此建议另外寻找交换功能的替代方案。你可以试着减少应用程序的内存使用量，并且确保任何时候都只加载让系统能够正常运行的一组最起码的二进制文件。

当然，如果存储设备是真正的硬盘（不是 CF 卡），那么交换功能便是一个可用的选项。然而启用交换功能将会导致响应时间变慢。

第八章

根文件系统的设置



准备好根文件系统的内容以及目标板的存储设备之后,接着就是设置可供目标板使用的根文件系统。首先我们需要为根文件系统选择文件系统的类型。然后我们需要将根文件系统的内容转换成选用的文件系统格式,或者将根文件系统安装在具备所选用文件系统格式的设备上。

本章首先会探讨选择文件系统的基本原则。接着会说明如何使用NFS将文件系统映像传送到目标板的flash中,这是本章常会用到的技术。然后我们针对CRAMFS、JFFS2、NFTL和RAM disk的使用探讨根文件系统的设置。最后我们会讨论到使用TMPFS安装特定的目录以及如何动态更新嵌入式系统的根文件系统。直到本章结束,还差bootloader(引导加载程序)的设置和配置没有讨论,我们就可以构建出一个全功能的嵌入式系统。我们将会在下一章探讨这些问题。

选择文件系统

为根文件系统选择文件系统是个困难的过程。最后通常是文件系统的能力与目标板的用途之间折衷的结果。举例来说,如果目标板根本不需要永久性地存储任何数据,那么选择可以提供永久性写入存储的文件系统毫无用处。对这类目标板来说,一个不具永久性存储功能的文件系统,例如CRAMFS,是比较好的选择。

此外,可能会考虑为同一个系统使用多个文件系统。例如,一个只需存取临时文件的系统,可以将根文件系统的大部分安装在CRAMFS之上,同时将`/var/tmp`目录安装在TMPFS或RAM disk上,`/tmp`是`/var/tmp`的符号链接。

描绘文件系统的特性

要为特定的应用选择最佳的文件系统或最佳的文件系统组合,我们需要一组可用来比较

文件系统的基本特性。表8-1整理列出了嵌入式Linux系统常会用到的文件系统特性。通过下面这几个问答题，可以描绘出每种文件系统的特性：

可被写入

这个文件系统可以被写入吗？

具有永久性

重引导之后这个文件系统可以保存修改过的内容吗？

具有断电可靠性

经变动的文件系统可以在断电之后恢复过来吗？

经过压缩

经安装的文件系统，其内容经过压缩吗？

存在RAM中

文件系统的内容在被安装之前会先从存储设备取出并放到RAM中吗？

表 8-1： 文件系统的特性

文件系统	可被写入	具有永久性	具有断电可靠性	经过压缩	存在于RAM中
CRAMFS	否	不适用	不适用	是	否
JFFS2	是	是	是	是	否
JFFS	是	是	是 ^a	否	否
NFTL 上的 Ext2	是	是	否	否	否
NFTL 上的 Ext3	是	是	是	否	否
RAM disk 上的 Ext2	是	否	否	否	是

a. 经过 Vipin Malik 的大量测试发现 JFFS 的断电可靠性可能会失败。然而，这类问题不会出现在 JFFS2 之上。详情参见 http://www.embeddedlinuxworks.com/articles/jffs_guide.html。

正如前面所说，一个系统只有在需要“更新文件系统上的数据”时，才需要使用一个可被写入的文件系统。同样地，一个系统只有在需要“重引导之后仍保存已更新的数据”时，才需要文件系统具备永久写入的功能。一个不提供写入能力的系统就不需要永久性存储以及断电可靠性的特性，因为它所存储的数据不会被修改。

另一方面，压缩是大多数文件系统渴望的特性，因为它可以提升嵌入式系统中存储设备的容量价格比，然而额外的压缩和解压缩工作却会降低 CPU 的性能。

大多数的文件系统都是从存放它们的存储设备直接安装，然而安装RAM disk 上的文件系统时，它们的内容在被安装之前必须先从存储设备取出并放到RAM中。因为你无法

直接存取存储设备上原本的文件系统映像，供RAM disk使用的文件系统映像在被放入存储设备之前多半经过压缩。我们将会“RAM disk上的磁盘文件系统”一节探讨如何建立此类经压缩的文件系统映像以提供RAM disk使用。

最后，无法更换已经被安装的文件系统。举例来说，如果一个系统的根文件系统从JFFS2格式的MTD分区安装，那么这个MTD分区的内容将无法被新的根文件系统覆盖。RAM disk是惟一可能的例外，因为文件系统被安装之前已经先复制到了RAM中，文件系统被安装之后，实际存放该文件系统的存储媒体并不会出现任何存取操作，因此可以放心地覆盖其内容。不过正如我们将在“在线更新”一节所见，一个以读写模式从原始存储设备安装的文件系统，仍旧可以通过各种方式来进行更新。

你可能已经注意到了，表8-1并未涵盖Linux支持的所有文件系统。因为表8-1只列出了适合嵌入式Linux系统使用的文件系统。此外，尽管表8-1提及JFFS，但是本章并没有加以探讨，这是因为它已经大量地被JFFS2所取代。

文件系统的选用原则

现在我们已经确立了一组用来描述文件系统的基本特性，接着让我们来探讨如何为MTD兼容存储设备选用文件系统。

ROMFS 与 ROMs 无关

你是否注意到内核配置菜单中File systems子菜单里的ROM file system support选项。这个文件系统实际上并未用到任何形式的ROM。它主要是将磁盘用于安装和调试的用途。ROMFS只能在块设备上运行，它无法以任何方式连接Linux的MTD子系统。正如计划网站上的说明，如果想让ROMFS使用真正的ROM，首先必须为这个ROM撰写设备驱动程序，使其成为块设备。详情参见<http://romfs.sourceforge.net/>。

如果系统只配备极少量的flash，但相对而言具有大量的RAM，那么RAM disk或许是最佳选择，因为RAM disk上的文件系统在存储设备上原本通常是经过压缩的。存储设备上供RAM disk使用的文件系统映像的压缩比通常比原生的压缩文件系统（例如CRAMFS或JFFS2）高很多，因为这类文件系统不会压缩它们的元数据（注1）。然而RAM disk的优势（存储设备上的文件系统经过压缩）却被较高的RAM使用量抵销，因为RAM中的文件系统未被压缩。此外，如果需要永久性存储，便不适合使用RAM disk。

注1：文件系统通常会使用此数据来确定文件和目录的位置以及维护文件系统的结构。

尽管如此，如果永久性存储需求有限，正如前面的提示，可以使用 RAM disk 来安装根文件系统的大部分内容，只有数据目录才从支持永久性存储的文件系统安装，例如 JFFS2。同时，使用 RAM disk 通常是建立自主目标板的最简单方式（目标板不需要通过主机取得它的内核或安装它的根文件系统）。举例来说，x86 系统（如 DAQ 模块）很可能会配备 RAM disk，因为这类系统的组件价格（包括 RAM）比其他架构都低。请注意，尽管 RAM 价格便宜，但它却比 flash 耗电。因此为 RAM disk 使用大量的 RAM 对于某些系统并不合适。

如果系统拥有稍微多一点的 flash，或者如果会尽可能给实际在目标板上执行的应用程序保留 RAM 空间，而且可以为运行时解压缩挪出若干额外的 CPU 周期，那么 CRAMFS 是个非常好的选项，只要系统符合“CRAMFS”一节提到的限制即可。尽管 CRAMFS 的压缩比低于 RAM disk（原因已经在前面说过了），不过它的能力对大多数不需要永久存储特性的嵌入式应用来说通常已经相当够用了。然而如同 RAM disk，可以在 CRAMFS 上安装根文件系统中不需于运行时变更的部分，以及在具有永久性的文件系统（例如 JFFS2）上安装其余的部分。

然而，如果目标板必须进行实地（在线）升级，CRAMFS 恐怕就不合适了。举例来说，iPAQ 的 Familiar 发行套件计划将 CRAMFS 换成了 JFFS2，因为前者让用户必须重新编程设备的 flash 才能更新它们的 iPAQ。另一方面来说，CRAMFS 却非常适合我的控制模块使用，因为在大多数的工业控制应用中实际的控制程序不会经常变动。

如果需要在任何时刻变更文件的任何部分，JFFS2 将会是最好的选择。尽管 JFFS2 的压缩比不如 CRAMFS，但因为 JFFS2 会为垃圾回收功能维护空间，并且它的元数据的结构允许文件系统的写入操作，所以 JFFS2 可以提供断电可靠性和损耗平衡这两个对使用 flash 系统而言非常重要的特性（参见第三章）。举例来说，我的用户接口模块便是完全以 JFFS2 来简化更新操作以及延长 flash 的寿命。然而在我写作本书时，如果使用的是 NAND 形式的 flash 设备，例如 DiskOnChip（DOC），便不适合使用 JFFS2（参见第三章）。

如果使用的是 DOC 设备，并且需要在任何时刻变更文件的任何部分，那么（在我写作本书时）只有“NFTL 上的磁盘文件系统”这个选项。配备 DOC 设备的嵌入式 x86 系统多半会使用这个配置。举例来说，我的 DAQ 模块被设成可以随时在当地把取样数据存入 NFTL 上的磁盘文件系统。

注意：严格地说，并不存在所谓的“磁盘文件系统”。然而我之所以会在本书使用这个术语，目的在于强调块设备上常用的文件系统（例如 ext2 和 reiserfs）与 MTD 设备上常用的文件系统（例如 JFFS2）之间的不同。

不管你使用的是 CRAMFS、JFFS2 还是 NFTL 上的磁盘文件系统，都可能需要安装 TMPFS 上的某些目录。尽管这个文件系统并不具备永久存储的特性，但是它却允许你使用部分的 RAM 来存储临时文件，功能如同根文件系统的 */tmp* 目录。如果使用的是以 CRAMFS 为基础的根文件系统，这让你能够拥有可以读写文件系统的一个目录或一对目录。如果使用的是 JFFS2 或是 NFTL 上的磁盘文件系统，这让你能够通过从 RAM 操作数据来避免存储设备的损耗。

以上所探讨的显然只是选用原则，而且你必须考虑每个系统的额外限制。尽管如此，这些指导原则代表建立嵌入式 Linux 系统时设计上所做的权衡，而且它们应该可以提供你如何做出最后设置的基本概念。接下来本章将会探讨前面所提到的文件系统的实际设置方式，并且会进一步探讨它们的使用细节。

磁盘设备的文件系统

如果想使用传统的磁盘作为系统的主要存储设备，例如系统通过 IDE 或 SCSI 接口连接的磁盘，建议参考 Linux 目前使用在工作站和服务器上的各种文件系统。尤其是，你会发现日志型文件系统（例如 ext3 和 reiserfs）相当适合应用在需要断电可靠性的环境里（例如嵌入式系统）。因为这些文件系统已经被广泛地应用在许多地方，而且在嵌入式系统上使用它们与在工作站或服务器上并无差别（注2），所以我将不会对它们的使用做进一步的探讨。不过你可以在探讨 Linux 在服务器和工作站上应用的典型文章中找到此类文件系统的详细说明。Daniel Robbins 为 IBM developerWorks 撰写的“Advanced filesystem implementor's guide”系列文章（<http://www.ibm.com/developerworks/linux/>）深入探讨了 Linux 的主要日志型文件系统，包括 ext3、reiserfs、JFS 和 XFS。也可以看一下 Derek Vadalà 所著的《Managing RAID on Linux》（O'Reilly）。

此外在《Running Linux》的第六章中也可以看到文件系统以工作站和服务器为方向的探讨方式。

使用经 NFS 安装的根文件系统将文件系统映像写入 flash 设备

我们会在第九章详细探讨如何在主机上进行 NFS 服务器的设置与配置以便为目标板提供根文件系统，现在让我们先看看该配置在这个阶段有什么用处。

注2：正如我们在第一章所做的讨论，嵌入式 Linux 系统的规模（CPU 的处理能力以及 RAM 的资源）已经足以容纳物理的硬盘。

在开发初期使用经NFS安装的根文件系统，可以通过快速修改目标板使用的文件来简化开发过程。接下来，为了让目标板能够独立运行，需要使用存放在flash设备上的文件系统。尽管有些bootloader可以用来将“文件系统映像”复制到flash设备，不过你还可以在目标板上执行MTD工具程序，从通过NFS安装的根文件系统，将“文件系统映像”复制到flash设备。要这么做，可以先把所选定的“文件系统映像”复制到用来存放“通过NFS安装的目标板的根文件系统”的目录，接着启动目标板，然后在目标板上使用MTD命令将“文件系统映像”复制到flash设备。

举例来说，如果要将最初的RAM disk映像复制到目标板的flash设备上，首先必须设定目标板的配置，让它能够从主机通过NFS导出的目录来安装根文件系统。接着必须在主机上将“文件系统映像”复制到用来导出给目标板的目录。尽管“文件系统映像”实际上并未放在目标板上，不过一旦内核在启动期间使用NFS安装该“文件系统映像”，它将会被视为目标板的根文件系统。然后启动目标板并且使用目标板上的MTD工具程序将文件系统映像从“通过NFS安装的根文件系统”复制到目标板的`/dev`目录里适当的flash设备条目中。

CRAMFS

CRAMFS是Linus Torvalds撰写的只具备最基本特性的文件系统。CRAMFS是个极简单（有时被过分简单化）、经压缩以及只读的文件系统，主要用于嵌入式系统。除了表8-1所列举的特性外，CRAMFS还具有以下限制：

- 每个文件最大不超过16 MB。
- 不提供“.”（当前）或“..”（上一级）目录条目。
- 文件的UID字段具有16位的宽度，GID字段具有8位的宽度。一般的文件系统通常会支持宽度16或32位的UID和GID字段。CRAMFS的GID字段被截短为较低的8位。换言之，建立在CRAMFS之上的根文件系统其GID字段的最大值为255（注3）。
- 所有文件的时间戳会被设成Unix epoch（即00:00:00 GMT, January 1, 1970）。尽管文件的时间戳可能会在运行时更新，不过更新的时间戳值将因inode在内存中的时间长短来定。一旦文件被重新载入之后，它的时间戳将会恢复到Unix epoch。
- 只有内存分页大小为4096字节的内核（PAGE_CACHE_SIZE的值必须设为4096）才可以读取CRAMFS的映像。

注3： 参见《Running Linux》第五章对UID和GID字段所做的说明。

- 不论是否经过链接，所有文件的链接计数（注4）皆为1。即使有多个文件系统项目指向同一个文件，该文件的链接计数也还是1。然而大部分的操作并不会因此受到影响，因为事实上你无法从 CRAMFS 移除任何文件。

如果目标板的根文件系统上的群组数不超过255，那么CRAMFS上被截短的GID字段并不会造成问题，如果目标板是一个单用户（single-user）系统，也不需要担心此项限制。如果系统必须支持多用户（multiuser）环境，请先确定所有文件和目录的GID字段皆低于255。否则，任何编号高于255的GID将会折回（wrap around）到低于255的数值，并且有可能造成安全上的问题。如果所使用的文件系统至少必须支持宽度16位的GID字段，那么你可能会想要在RAM disk上使用磁盘文件系统。它可以像CRAMFS那样在存储媒体上提供压缩的功能，也可以提供读写的存取功能，然而CRAMFS只能提供读取的存取功能。

除了CRAMFS的限制，用来为CRAMFS文件系统建立映像的工具会受到主机的字节顺序的影响。因此你用来建立CRAMFS映像的主机必须具备跟目标板一样的字节顺序。越过此限制的惟一方法就是使用我在上一节提到的技术。基本上，必须以NFS来安装目标板的根文件系统，为目标板在经NFS安装的文件系统上建立CRAMFS映像，并且将建立的CRAMFS映像写入flash设备。尽管撰写本书时内核源码树中的CRAMFS建立工具仍然存在这项限制，不过你可以通过修补最新版的CRAMFS工具来得到与主机的字节顺序无关的文件系统建立工具。你可以在<http://sourceforge.net/projects/cramfs/>上找到最新版的CRAMFS工具套件，该网站的Patches专区有交换字节顺序的补丁。

如果系统符合CRAMFS的限制条件，你或许会考虑在自己的计划中使用CRAMFS。如果想要使用CRAMFS，但系统不符合它的限制条件，你可能会想要使用CRAMFS经过修改的版本。正如之前所提到的主机字节顺序的问题，人们会通过修改CRAMFS来克服这些限制条件。举例来说，第三章述及的linuxppc-embedded邮件论坛中便出现若干报告提到有人通过修改CRAMFS来避免页大小的问题。尽管这类修补并不属于主流内核源码树中CRAMFS程序代码的一部分，不过你会发现它们很有用。你可以在前面提到的补丁专区找到常见的CRAMFS补丁。

要为根文件系统建立CRAMFS映像，首先需要建立并安装CRAMFS工具：*cramfsck*和*mkcramfs*。这两个工具程序是计划网站发行套件的一部分，可以在（2.4.18版）内核源码树的*scripts/cramfs*目录里找到它们的程序代码。要从内核源码树中建立这两个工具程序，请移往*scripts/cramfs*目录并执行*make*命令：

注4：如同其他的Unix系统，也可以为大部分的Linux文件系统建立指向文件的链接。通常，这类文件系统会为每个文件计算指向它的链接数目，当此计数为0时，相应的文件便会被删除。


```
$ cd ${PRJROOT}/kernel/linux-2.4.18/scripts/cramfs
$ make
```

接着，将这两个工具复制到适当的目录：

```
$ cp cramfsck mkcramfs ${PREFIX}/bin/
```

然后，可以为目标板的根文件系统建立 CRAMFS 映像：

```
$ cd ${PRJROOT}
$ mkcramfs rootfs/ images/cramfs.img
bin
boot
dev
etc
lib
linuxrc
proc
sbin
tmp
usr
'bin':
  addgroup
...
'boot':
  boot.b
...
'sbin':
  chroot
Directory data: 6484 bytes
166.67% (+15 bytes)    addgroup
-31.46% (-2196 bytes) allinone
-40.27% (-240 bytes)  arch
185.71% (+13 bytes)   ash
...
-49.60% (-3700 bytes) wall
-49.54% (-695 bytes)  include
Everything: 3560 kilobytes
Super block: 76 bytes
CRC: f18594b6
warning: gids truncated to 8 bits. (This may be a security concern.)
```

此例中，*rootfs/* 占用 7840 KB 的存储空间，然而 CRAMFS 映像的大小只有 3560 KB，大约有 50% 的压缩比。此值与 CRAMFS 的典型压缩比一致。

准备好文件系统映像后，接着可以将它写入存储设备：

```
$ su -m
Password:
# cat rootfs/cramfs.img > /dev/mtd4
# exit
```

当然，以上的命令假设主机可以存取该存储设备。如果不行，首先必须按照上一节的说明使用经NFS安装的根文件系统。并使用前面建立的*cramfsck* 工具程序来检查CRAMFS文件系统的内容。

JFFS2

我已经在第三章探讨过了JFFS2的特性。不过我尚未提到JFFS2性能的一项限制。因为它的架构，JFFS2在MTD扇区上实现了“垃圾收集”功能。启用这项功能通常不会有什么问题。然而，当文件系统达到它的极限，JFFS2花费在“垃圾收集”的时间将会变大。此外，当文件系统达到它的极限，系统将无法删除或移动文件，而且存取文件的速度会变慢。如果使用的是JFFS2，确定应用程序的数据不会增长到填满整个文件系统。换言之，也就是确定应用程序在将数据写入文件系统之前会检查文件系统的可用空间，以避免严重降低系统的速度，进而造成系统崩溃。此外，还可以在目标板上进行“基准”测试，以便决定JFFS2运行开始失常的临界值。

知道该项限制之后，现在让我们专注于JFFS2文件系统映像的建立和安装。我们将会使用前一章在进行MTD工具程序的安装时安装的*mkfs.jffs2* 工具程序。

JFFS2映像的建立相当简单：

```
$ cd ${PRJROOT}
$ mkfs.jffs2 -r rootfs/ -o images/rootfs-jffs2.img
```

我们将会使用*-r*选项来指定内含根文件系统的目录，并使用*-o*选项来指定文件系统映像的输出文件名称。除了这两个选项，我们还可以分别使用*-l*或*-b*来建立字节顺序为小尾端（little endian）或大尾端（big endian）的映像。JFFS2的“压缩比”比CRAMFS小很多。举例来说，一个7840 KB的根文件系统，建成JFFS2映像之后它的大小仍有6850 KB。其“压缩比”仅比10%高一点。

建好JFFS2映像之后，可以将它写入选定的MTD设备。如果主机可以存取该设备，可以直接在主机上执行适当的命令。否则请按照上一节的指示：开启经NFS安装根文件系统的目标板，将JFFS2映像摆在文件系统中，并在目标板上执行命令将映像写入选定的MTD设备。不管设置怎样，写入映像之前，首先需要抹除MTD设备：

```
# eraseall /dev/mtd5
Erased 8192 Kibyte @ 0 -- 100% complete.
```

显然，MTD存储设备上的可用空间必须等于或大于你要写入的JFFS2映像。抹除MTD设备之后，可以将JFFS2映像复制到MTD分区：

```
# cat images/rootfs-jffs2.img > /dev/mtd5
```

接着安装你所复制的文件系统，然后检查一番：

```
# mount -t jffs2 /dev/mtdblock5 /mnt
# mount
...
/dev/mtdblock5 on /mnt type jffs2 (rw)
# ls /mnt
bin      etc      linuxrc  sbin     usr
dev      lib      proc     tmp      var
# umount /mnt
```

与磁盘文件系统不同的是，无法在 loopback 设备上使用 *mount -o loop ...* 命令安装 JFFS2。你必须从真正的 MTD 设备来安装 JFFS2，正如上面的范例所示。如果主机上没有真正的 MTD 设备可用，例如 CFI flash，那么你可以使用第三章提到的虚拟存储 MTD 设备。你还可以使用前一章提到的 *jffs2reader* 命令来检查映像的内容。

如果目标板先前使用的是经 NFS 安装的根文件系统，那么你现在可以启动目标板，并把 JFFS2 文件系统作为它的根文件系统。

NFTL 上的磁盘文件系统

我们已经在前一章讨论过了 NFTL 与 DOC 设备的安装与使用。现在让我们来探讨如何使用 NFT 仿真的块设备上的磁盘文件系统。ext2 是 NFTL 上使用得最广泛的磁盘文件系统。因此我们将会专门探讨这个特例。然而请注意，NFTL 上的 ext2 并不提供“断电可靠性”。如果有这方面的需求，应该在 NFTL 上使用日志型文件系统，例如 ext3、XFS、JFS 或 reiserfs。你可以针对任何既有的磁盘文件系统，包括日志型文件系统，轻易改变接下来的命令。

首先在特定的 NFTL 分区上为选定的文件系统类型建立文件系统：

```
# mke2fs /dev/nftl1a1
mke2fs 1.18, 11-Nov-1999 for EXT2 FS 0.5b, 95/08/09
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
8160 inodes, 32574 blocks
1628 blocks (5.00%) reserved for the super user
First data block=1
4 block groups
8192 blocks per group, 8192 fragments per group
2040 inodes per group
Superblock backups stored on blocks:
    8193, 24577
```

```
Writing inode tables: done
Writing superblocks and filesystem accounting information: done
```

接着安装分区并且将根文件系统复制过去：

```
# mkdir /mnt/doc
# mount -t ext2 /dev/nftla1 /mnt/doc
# cp -a rootfs/* /mnt/doc
```

我在此处会假定你是从项目的 `$(PRJROOT)` 目录下执行这些命令的。我还会假定你可以在主机上使用 `/dev/nftla` 来存取 DOC 设备，而且你想要在该设备的第一分区上建立 ext2 文件系统。如果主机无法存取 DOC 设备，请按照上一节的说明，使用经 NFS 安装的根文件系统把根文件系统的内容复制到 DOC 设备。

RAM disk 上的磁盘文件系统

RAM disk 正如其名，存在于 RAM 中并且功能犹如块设备。内核可以在同一时间支持多个活动的 RAM disk。因为它们的功能犹如块设备，所以 RAM disk 上可以使用任何磁盘文件系统。由于 RAM disk 上的内容将因系统的重新开机而丢失，所以 RAM disk 通常会从经压缩的磁盘文件系统（例如 ext2）加载其内容，这就是所谓的经压缩的 RAM disk 映像。这类经压缩的 RAM disk 映像特别适合在嵌入式 Linux 系统系统初始化期间应用。换言之，内核必须具备从存储设备取出 initrd (initial RAM disk) 映像作为它的根文件系统的能力。启动时，内核会确认引导选项是否有指示 initrd 的存在。如果有，内核会从所选定的存储设备取出文件系统映像（不论它是否经过压缩）放入 RAM disk，并且将它安装成根文件系统。事实上，initrd 机制是为内核提供根文件系统的最简单方法。本节中，我们将会探讨如何建立经压缩的 RAM disk 映像用作 initrd。我将会在第九章说明此映像实际上如何用作 initrd。

根据设置，我们将会建立以 ext2 为基础的 RAM disk 映像供目标板使用。尽管 ext2 是 RAM disk 最常用的文件系统，不过你也可以使用其他的磁盘文件系统，正如我在之前提示的一样。例如有些开发者表示他们会使用 CRAMFS。

请注意，尽管在接下来的程序中我们将会建立文件系统映像供 RAM disk 使用，不过所有的操作都会在主机的磁盘上进行。因此以下的步骤并不会涉及如何在主机上使用实际的 RAM disk。

首先为根文件系统建立一个空的文件系统映像：

```
$ cd $(PRJROOT)
$ mkdir tmp/initrd
$ dd if=/dev/zero of=images/initrd.img bs=1k count=8192
```

```
8192+0 records in
8192+0 records out
```

此处以 *dd* 命令建立了一个 8192 KB 的文件系统映像，并以 */dev/zero* 对它进行初始化。用这种方式对文件系统进行初始化，稍后当我们使用 *gzip* 压缩整个映像时，将让文件系统中未使用的部分获得最大的压缩比。相对而言，如果我们使用的是与前面建立的映像相同大小的现存映像而不使用 *dd* 命令，假定其中已经包含不一致的数据，则会产生较低的压缩比。事实上，这代表你不应该更新当前的文件系统映像。取而代之的是，总是建立新的文件系统映像，以便使用目标板已更新的根文件系统目录 *\$(PRJROOT)/rootfs*。

对文件系统映像进行初始化之后，接着对它建立文件系统并且安装它：

```
$ su -m
Password:
# /sbin/mke2fs -F -v -m0 images/initrd.img
mke2fs 1.18, 11-Nov-1999 for EXT2 FS 0.5b, 95/08/09
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
2048 inodes, 8192 blocks
0 blocks (0.00%) reserved for the super user
First data block=1
1 block group
8192 blocks per group, 8192 fragments per group
2048 inodes per group
Writing inode tables: done
Writing superblocks and filesystem accounting information: done
# mount -o loop images/initrd.img tmp/initrd
```

此处，*-F* 选项将可迫使 *mke2fs* 在文件上运行。否则，*mke2fs* 会抱怨 *images/initrd.img* 不是个块设备。*-v* 选项指出 *mke2fs* 应该以 *verbose* 模式执行，*-m0* 指出不必在文件系统上为“超级用户”保留任何区块。当你为工作站或服务器的使用建立文件系统时，为“超级用户”保留区块有其实质上的意义，然而在嵌入式系统中却不是很有用，因为它们通常会被建立成单用户系统。

现在将根文件系统复制到 RAM disk 并且卸载它：

```
# cp -av rootfs/* tmp/initrd
rootfs/bin -> tmp/initrd/bin
rootfs/bin/busybox -> tmp/initrd/bin/busybox
rootfs/bin/ash -> tmp/initrd/bin/ash
rootfs/bin/cat -> tmp/initrd/bin/cat
rootfs/bin/chgrp -> tmp/initrd/bin/chgrp
rootfs/bin/chmod -> tmp/initrd/bin/chmod
...
# umount tmp/initrd
# exit
```

第一个命令执行后，将会看到根文件系统中所有文件的完整列表，如上例所示你会看到这些文件的完整路径名称。现在 *images/initrd.img* 文件中包含了目标板的整个根文件系统。最后一个步骤就是压缩文件系统以便产生一个经压缩的 RAM disk:

```
$ gzip -9 < images/initrd.img > images/initrd.bin
$ ls -al images/initrd*
-rw-rw-r-- 1 karim karim 3101646 Aug 16 14:47 images/initrd.bin
-rw-rw-r-- 1 karim karim 8388608 Aug 16 14:46 images/initrd.img
```

使用 *gzip* 命令压缩文件系统，-9 选项用来告诉此命令使用最高级的压缩算法。这将获得优于 CRAMFS 和 JFFS2 的压缩比，大约 60%。然而，就 RAM disk 存在于 RAM 中这个事实来说，这个结果受到“选择文件系统”一节提及的限制。

你可以将此处建立的 RAM disk 映像，*images/initrd.bin*，放到目标板中适当的设备上，并据此设定 bootloader 的配置。正如前文所述，我们将会在第 9 章探讨如何将 RAM disk 当成 initrd 来用。

init RAMFS

写作本书时，尽管 initrd 仍是常用的方法，不过 initrd 机制已经渐显老态，于是内核开发者有意在 2.5 版内核系列中用 *init RAMFS* (initramfs) 取代它。未来每个内核都有可能在 initramfs 映像中纳入目前硬编码在内核里的初始化程序代码。这样，一旦初始化完成之后，便可使用 `pivot_root()` 系统调用，用 initramfs 上执行的最后程序来安装最后的根文件系统。在主流版本采用 initramfs 之前，initrd 仍是开机时为内核提供根文件系统的标准方法。

安装 TMPFS 上的目录

TMPFS 是一个以虚拟内存为基础的文件系统，它的大小可以根据实际的内容缩放。尽管它的内容会在重新开机后丢失，不过它相当适合用来存储临时文件。因此，除了可以使用单一文件系统来安装所有的目录外，也可以选择在 TMPFS 上安装不需要永久存储的目录，例如 */tmp*。然而，因为 TMPFS 上存放的内容将在重新开机后丢失，所以一些必要的目录，例如 */usr*、*/etc* 或 */bin*，无法存放在 TMPFS 上。要使用 TMPFS，请在内核配置菜单的 File systems 子菜单中启用“Virtual memory file system support (former shm fs)”的支持。

在内核启用 TMPFS 的支持后，举例来说，可以在 */tmp* 目录上安装 4 MB 的 TMPFS 文件系统：

```
# mount -t tmpfs none /tmp -o size=4m
```

此外，可以在 */etc/fstab* 文件中加入相应的设定，并且修改 */etc/init.d/rcS* 文件，以便在引导期间安装 TMPFS。如果并未指出大小的限制，文件系统将会随着内容的增加而长大。

与大多数其他的 *mount* 命令相比，TMPFS 不必指出所要安装的设备或文件，因此使用 *none* 来作为这个设备的名称。*mount* 实际上会忽略 TMPFS 的这个设备名称，并且会将 *none* 替换成对命令没有影响的任何其他名称。

如果想要了解 TMPFS 的相关细节，可参考前面提到的 IBM developerWorks 的“Advanced filesystem implementor's guide”系列文章的第 3 部分“Using the virtual memory (VM) filesystem and bind mounts”(<http://www-106.ibm.com/developerworks/library/l-fs3.html>)。

在线更新

正如本章前面所说，当你从存储媒体安装文件系统之后，便无法全面取代整个文件系统。因此我们必须寻找其他的方法来更新已安装的文件系统的内容。完成此任务的方法很多，每个方法各有其优缺点。本节将会探讨三种方法：*rsync* 工具程序、套件管理工具、特殊的命令脚本。

rsync 工具程序

rsync 是个远程更新工具程序，它让本地的目录树与远程服务器同步。它只会传送本地和远程文件之间的不同之处。它会保存文件存取权、文件所有权、符号链接、存取时间以及设备条目。*rsync* 可以使用 *rsh* 或 *ssh* 来与远程服务器通信。因为具有这些特性，所以 *rsync* 适合用来更新具备联网功能的嵌入式系统。在 *rsync* 计划位于 <http://samba.anu.edu.au/rsync/> 的网站上可以找到 *rsync* 套件、相关文档以及邮件论坛。除了计划网站提供的文档，Michael Holve 于 <http://everythinglinux.org/rsync/> 提供的教学文档也非常值得一读。

要使用 *rsync*，服务器上必须执行 *rsync* 监控程序，嵌入式系统则必须执行 *rsync* 客户程序。我将不会提到如何安装 *rsync* 服务器，也不会说明 *rsync* 客户端的使用细节，因为前面提到的教学文档以及 *rsync* 本身的文档已经涵盖了这方面的内容。尽管如此，我将会说明如何交叉编译及安装 *rsync* 以供目标板使用。

首先下载 *rsync* 套件并且将源码取出放到 `${PRJROOT}/sysapps` 目录下。例如，我的 UI 模块使用的是 2.5.6 版的 *rsync* 套件，取出源码之后，我将会移往存放源码的目录，以便进行其余的操作：

```
$ cd ${PRJROOT}/sysapps/rsync-2.5.6/
```

接着设定并编译此套件：

```
$ CC=arm-linux-gcc CPPFLAGS="-DHAVE_GETTIMEOFDAY_TZ=1" ./configure \
> --host=$TARGET --prefix=${TARGET_PREFIX}
$ make
```

若要使用 *uClibc*（而不要使用 *glibc*）进行编译，可将 *arm-linux-gcc* 替换成 *arm-uclibc-gcc*。此处，我们必须设定 *CPPFLAGS* 来将 *HAVE_GETTIMEOFDAY_TZ* 定义为 1，否则会造成编译失败，因为 *configure* 命令脚本无法正确决定目标板的 *gettimeofday()* 会使用多少个参数。

编译完成之后，接着将 *rsync* 二进制文件安装到目标板的根文件系统并且对它进行 *strip* 的处理：

```
$ cp rsync ${PRJROOT}/rootfs/bin
$ arm-linux-strip ${PRJROOT}/rootfs/bin/rsync
```

若二进制文件以动态方式链接 *uClibc* 或 *glibc*，则经 *strip* 处理的二进制文件大小会有 185 KB。若二进制文件以静态方式链接 *uClibc*，则经 *strip* 处理的二进制文件大小会有 270 KB；若二进制文件以静态方式链接 *glibc*，则经 *strip* 处理的二进制文件大小会有 655 KB。

同一个二进制文件可以在命令行上使用，也可以当成监控程序来用。*--daemon* 选项用来要求 *rsync* 以监控程序的方式执行。就我们的例子而言，我们将只会在命令行使用 *rsync*。要使用 *rsync*，必须事先为目标板安装 *rsh* 或 *ssh*。*rsh* 是 *netkit-rsh* 套件的一部分（可从 <ftp://ftp.uk.linux.org/pub/linux/Networking/netkit/> 取得）。*ssh* 则是 *OpenSSH* 套件的一部分，我们将会在第 10 章对 *OpenSSH* 套件做深入的探讨。尽管该项讨论专注在如何使用 *OpenSSH* 产生的 *SSH daemon (sshd)*，不过 *OpenSSH* 套件在编译期间还会产生 *SSH 用户程序*。接下来，我会假定你使用的是 *ssh* 而不是 *rsh*，因为前者可以提供安全的通信管道。然而使用 *ssh* 的缺点就是文件太大，动态链接的 *SSH 用户程序* 经 *strip* 处理后约有 1.1 MB，静态链接时就更大了。相对而言，动态链接的 *rsh* 经 *strip* 处理后只有 8 KB。

rsync 安装到目标板上后，可以在目标上使用如下的命令来更新它的根文件系统：

```
# rsync -e "ssh -l root" -r -l -p -t -D -v --progress \
> 192.168.172.50:/home/karim/control-project/user-interface/rootfs/* /
root@192.168.172.50's password:
```



```
receiving file list ... done
bin/
dev/
etc/
lib/
sbin/
tmp/
usr/bin/
usr/sbin/
bin/busybox
750756 (100%)
bin/tinylogin
39528 (100%)
etc/inittab
377 (100%)
etc/profile
58 (100%)
lib/ld-2.2.1.so
111160 (100%)
lib/libc-2.2.1.so
1242208 (100%)
...
sbin/nftl_format
8288 (100%)
sbin/nftldump
7308 (100%)
sbin/unlock
3648 (100%)
bin/
dev/
etc/
lib/
sbin/
wrote 32540 bytes  read 2144597 bytes  150147.38 bytes/sec
total size is 3478029  speedup is 1.60
```

这个命令会从我的主机（位于IP地址192.168.172.50）将“UI模块”项目工作空间中的 *rootfs* 目录的内容复制到目标板的根目录。为了让这个命令执行成功，我的主机必须执行 *sshd* 和 *rsync* 这两个监控程序。

此外，还需要以下选项：

- e 将用来连接远程服务器的应用程序的名称传给 *rsync*。（此例中，我们会使用 *ssh -l root* 以 *root* 身份连接服务器。你也可以将 *root* 替换成任何最适当的用户名称。如果未提供用户名称，*ssh* 将会以会话拥有者的用户名称来连接服务器。）
- r 以递归的方式复制目录。
- l 保存符号链接。
- p 保存文件存取权限。

- t 保存时间戳。
- D 保存设备节点。
- v 提供详细的输出。
- progress
报告传输的过程。

执行时, *rsync* 会为每个文件或目录副本提供一份清单, 并且维护一个计数器以便显示传输已经完成的百分比。接着 *rsync* 会复制远程目录, 于是目标板的根文件系统将会跟服务器上最新的目录内容同步。

如果只想检查哪些文件将被更新而不想执行实际的更新动作, 可以使用 *-n* 选项要求 *rsync* 进行“排练”(dry run):

```
# rsync -e "ssh -l root" -r -l -p -t -D -v --progress -n \  
> 192.168.172.50:/home/karim/control-project/user-interface/rootfs/* /  
root@192.168.172.50's password:  
receiving file list ... done  
bin/busybox  
bin/tinylogin  
etc/inittab  
etc/profile  
lib/ld-2.2.1.so  
lib/libc-2.2.1.so  
...  
sbin/nftl_format  
sbin/nftldump  
sbin/unlock  
wrote 176 bytes  read 5198 bytes  716.53 bytes/sec  
total size is 3478029  speedup is 647.20
```

关于如何使用 *rsync* 的进一步信息 (包括客户端和服务端), 可参考该命令的 manpage 以及计划网站提供的说明文档。

套件管理工具

事实上, 并不可能或需要每次都像前一节那样, 使用 *rsync* 同时更新构成根文件系统的所有软件套件。有时最好的做法就是工作站和服务端发行套件中常见的做法: 使用套件管理系统分别更新每个套件。举例来说, 如果工作站使用的是 Linux, 你或许已经熟悉 Linux 上两种主要的套件管理系统其中一个: *RPM Package Manager* (RPM) 或 *Debian package* (dpkg)。因为这些系统的完善记录可以协助用户和系统管理员维护系统的最新及最佳运行状态, 所以值得在嵌入式系统中交叉编译这些系统的工具。然而, 这两个系统对系统资源的要求很高, 因此不适合直接在嵌入式系统中使用。

幸好，目前存在以嵌入式系统为目标的工具，让我们不必耗费太多系统资源就能获得套件管理工具的强大功能。它们是 BusyBox 的 *dpkg* 命令以及 *Itsy Package Management System* (iPKG)。BusyBox 的 *dpkg* 命令可让我们在嵌入式系统中安装 *dpkg* 套件。与其他的 BusyBox 命令非常像的是，可以将它设成 *busybox* 二进制文件的一部分。iPKG 则是本书前面提到的 Familiar 发行套件使用的套件管理系统。你可到该计划位于 <http://www.handhelds.org/z/wiki/iPKG> 的网站取得该套件以及使用说明文件。尽管 iPKG 有自己的格式，不过它也可以处理 *dpkg* 套件。

iPKG 套件的建立说明可以在 <http://www.handhelds.org/z/wiki/BuildingIpkg> 上找到。*dpkg* 套件的建立说明可以参考“Debian New Maintainers' Guide”和“Dpkg Internals Manual”（可至 <http://www.debian.org/doc/devel-manuals> 查看）。BusyBox 的 *dpkg* 命令的使用说明可参考 BusyBox 的文件，*ipkg* 工具则包含在 iPKG 套件管理系统中，它的使用说明可以在该计划的网站上找到。

特殊的命令脚本

如果因为某种理由，前面提到的工具无法进行嵌入式系统的根文件系统的更新工作，我们仍然可以使用更基本的文件处理工具程序来进行更新的工作。基本上，我们不是使用 *cp* 命令来复制每个文件，就是使用 *patch* 命令来修补一组文件，或是此二者同时进行。不管如何做，我们都需要在主机上把改变的部分包装成套件，并且在目标板上应用这些变更套件。最简单的方法就是使用 shell 命令脚本来建立并应用变更套件。

diff 与 patch

尽管 *diff* 与 *patch* 可用来修补整个目录结构，不过它们会把符号链接当成一般文件来处理，所以最后的结果是复制被链接文件的内容，而不是建立符号链接。因此，*diff -aurN oldrootfs/rootfs/* 建立的补丁毫无用处。这两个套件的未来计划之一就是修改工具程序，让它们能够正确处理符号链接。

建立这类命令脚本时，必须考虑到文件间的依存关系。举例来说，如果要更新一个链接库，我们必须确定文件系统上与该链接库有依存关系的链接库，是否在使用新版链接库的情况下仍能正常运行。例如 0.9.15 版的 uClibc 变更了二进制文件的格式。因此，如果 uClibc 更新为 0.9.15 或之后的版本，则任何跟 0.9.14 和之前版本的 uClibc 链接的应用程序都必须更新。尽管这类变动并不常见，不过仍必须小心应对。一般，任何涉及链接库的更新工作都必须小心处理，以避免造成系统失效。欲进一步了解如何正确地更新链接库，可参考《Running Linux》第七章的“Upgrading Libraries”这一节。

安装 patch 工具程序

建立更新命令脚本的第一步，就是在主机和目标板上有适当的工具可用。因为 *diff* 和 *patch* 通常已经安装在主机上，所以让我们来探讨如何为目标板安装 *patch*。

要将 *patch* 安装到目标板的根文件系统，首先得从 GNU 计划位于 *ftp://ftp.gnu.org/gnu/patch/* 的 FTP 网站下载 GNU *patch* 工具程序。例如我的 UI 模块使用的是 2.5.4 版的 *patch*。套件下载之后，接着取出源码并放到 *\${PRJROOT}/sysapps* 目录下。

现在为工具程序产生建立目录：

```
$ cd ${PRJROOT}/sysapps
$ mkdir build-patch
$ cd build-patch
```

接着设定、建立和安装套件：

```
$ CC=arm-uclibc-gcc ../patch-2.5.4/configure --host=$TARGET \
> --prefix=${TARGET_PREFIX}
$ make LDFLAGS="-static"
$ make install
```

请注意，我们在此处使用 *uClibc*，并且静态链接命令。事实上，我们还可以使用 *glibc* 或 *diet libc*。然而不管使用何种链接库，以静态方式链接 *patch* 的好处是，不会因为更新期间造成链接库丢失或不完整而导致 *patch* 在目标板上执行失败。

将 *patch* 工具程序安装到 *\${TARGET_PREFIX}/bin* 之后，可以接着从该目录把它复制到目标板根文件系统的 */bin* 目录，供目标板使用。一旦 *patch* 放到目标板的根文件系统之后，可以使用适当的 *strip* 命令来缩减工具程序的大小。例如下面是我为 UI 模块安装 *patch* 的过程：

```
$ cp ${TARGET_PREFIX}/bin/patch ${PRJROOT}/rootfs/bin
$ cd ${PRJROOT}/rootfs/bin
$ ls -al patch
-rwxrwxr-x    1 karim    karim        252094 Sep  5 16:23 patch
$ arm-linux-strip patch
$ ls -al patch
-rwxrwxr-x    1 karim    karim        113916 Sep  5 16:23 patch
```

进行更新的命令脚本

下面这个 shell 命令脚本是根据前面提到的目标板更新原则设计的，它可以在主机上建立用来更新目标板根文件系统的套件：

```
#!/bin/sh
```

```
# 文件名: createupdate
# 参数 $1: 存放原始根文件系统的目录
# 参数 $2: 存放已更新之根文件系统的目录
# 参数 $3: 用来存放修补文件和更新文件的目录
# 参数 $4: 欲更新的uClibc 链接库

# 比较/etc 目录
diff -urN $1/etc $2/etc > $3/etc.diff

# 复制BusyBox和TinyLogin
cp $2/bin/busybox $2/bin/tinylogin $3/

# 复制uClibc 组件
cp $2/lib/*$4* $3
```

这个命令脚本做了几项假设。首先它假设/etc 目录以及它的任何子目录并未包含符号链接。即使包含了符号链接，我们仍然可以用 -x 或 -X 选项将符号链接排除在外。此外，这个命令脚本还会更新 BusyBox、TinyLogin 和 uClibc。你可以根据自己的设置加入适当的 cp 和 diff 命令。

以下是这个命令脚本的使用范例：

```
$ cd ${PRJROOT}
$ mkdir tmp/rootfsupdate
$ createupdate oldrootfs/ rootfs/ tmp/rootfsupdate/ 0.9.14
```

此例中，oldrootfs 包含了目标板上目前的根文件系统，rootfs 包含了根文件系统的最新版本，tmp/rootfsupdate 包含了用来更新目标板的文件和补丁，并且新的 uClibc 版本为 0.9.14。

接下来的命令脚本将会使用前面所建立的“更新目录”来更新目标板：

```
#!/bin/sh

# 文件名: applyupdate
# 参数 $1: 存放修补文件和更新文件目录的绝对路径
# 参数 $2: 旧的uClibc 版本
# 参数 $3: 新的uClibc 版本

# 修补/etc
patch -p1 < $1/etc.diff

# 复制BusyBox和TinyLogin
cp $1/busybox $1/tinylogin /bin/

# 复制欲更新的uClibc 组件
cp $1/*$3* /lib

# 更新uClibc 的符号链接
ln -sf libuClibc-$3.so /lib/libc.so.0
for file in ld-uClibc libcrypt libdl libm libpthread libresolv libutil
```

```
do
ln -sf $file-$3.so /lib/$file.so.0
done

# 移除旧的uClibc 组件
rm -rf /lib/*$2*
```

这个命令脚本比用来建立“更新目录”的命令脚本长一点。之所以会比较复杂是因为它必须进行 C 链接库组件替换工作。请注意，此处我们会使用 *ln -sf*（不会先删除链接再使用 *ln -s*）来进行这个工作。这件事很重要，因为完全删除链接会造成系统无法使用。接着你必须将目标板关机，并且以适当的方式重新编程它的存储设备。

要执行这个命令脚本，先将 *rootfsupdate* 目录复制到目标板的 */tmp* 目录，再执行该命令行：

```
# applyupdate /tmp/rootfsupdate 0.9.13 0.9.14
```

实际在目标板上使用这个命令行之前，可以先在主机上测试它。以下是测试的步骤：

1. 从 */\${PRJROOT}* 目录将旧的根文件系统（可能是 *oldrootfs*）复制到 *tmp*。
2. 修改命令脚本，移除绝对路径中开头的“/”符号。例如，将 */etc* 替换成 *etc*。
3. 在欲复制的文件系统上执行此命令脚本。
4. 手动检查是否顺利完成更新的动作。

使用 GNU cp 以外的方式来复制整个目录树

建立嵌入式 Linux 系统的时候，通常需要从一個位置将整个目录复制到另一个位置，并且尽可能保持文件、目录和符号链接的原样。在我前面的说明中已经这么做过几次了，并且反复使用 *cp -a* 命令完成此事。尽管 GNU *cp* 已经支持 *-a* 选项一段时间了，但如果使用的不是 Linux，那么系统上可能没有安装 GNU *cp*。如果因为某个理由系统并未安装 GNU *cp*，仍然可以使用一个由 *cd* 和 *tar* 组合的命令来获得与 *cp -a* 相同的结果。让我们进一步检查这个命令的工作原理。下面是这个命令的一般形式：

```
$ (cd SRC_DIR && tar cf - .) | (cd DEST_DIR && tar xvf -)
```

这个命令分成两个部分。“|”符号左边的命令会移往 *SRC_DIR* 目录并在该目录中启动 *tar*。其中，*tar* 命令会被要求把本地目录的内容建成 *tar* 包文件并且将所产生的包文件转储（dump）到标准输出。在 *tar* 建立包文件的简单用法中，命令后面会

—待续—

跟着大于号(>)以及磁带机或磁盘文件的名称。此处，我们并不会存储输出结果；我们只是利用 *tar* 将文件放入流中，以便送往其他地方。

在 Unix 用来执行命令的 shell 上，“|”符号会为左边命令的输出与右边命令的输入之间建立一个管道(pipe)。因此左边命令的标准输出上所转储(dump)的包文件会被提供给右边命令的标准输入。接着“|”符号右边的命令会移往 *DEST_DIR* 目录并且在该目录中启动 *tar*。与第一个 *tar* 命令相比，第二个 *tar* 命令会将标准输入里的内容取出并放到本地目录。

这个命令的最终结果是将 *SRC_DIR* 目录里的文件和目录原封不动地复制到 *DEST_DIR* 目录。因此这个命令执行之后，*DEST_DIR* 目录的内容跟 *SRC_DIR* 目录一样。

如果系统上安装了 GNU *cp*，这个命令就没有多少用处，但如果系统上没有 GNU *cp*，你可能会发现这个命令很有用。如果使用的是标准的 Linux 工作站或服务器发行套件，则 *cp -a* 命令仍然是复制整个目录树较好的选择。

第九章

设置 bootloader



bootloader 主要负责加载内核，尽管它在系统启动期间执行的时间非常短，不过它却是非常重要的系统组件。在一定程度上，设置 bootloader 是所有 Linux 系统的一项常见工作。尽管如此，对嵌入式系统来说，这却是一件特别的工作，因为将 bootloader 用在这类系统中，可能完全不同于将它们用在一般的系统中，即使相同，它们的配置和操作方式也可能相去甚远。

在第七章探讨过了嵌入式存储设备的操作，并在第8章说明过了如何为目标板设置根文件系统之后。现在要来探讨如何设置 bootloader，再加上前面建立的其他组件，我们将可以获得一个可引导且全功能的嵌入式系统。因为不同的硬件架构以及基于相同架构的不同电路板之间有很大的差异，所以 bootloader 的选用、设置以及配置跟所使用的硬件有很大的关系。

因为 Linux 有许多 bootloader 可用，再加上有成千上万的嵌入式电路板以及相同的电路板有各种可能的引导配置，因此单凭一章的篇幅不足以涵盖所有可能的组合，也不可能深入探讨所提到的每个 bootloader 的用法。Linux 许多当前的 bootloader 不是已经有专书探讨它们的用法，就是需要为它们撰写一本书。

此外，不同架构之间 bootloader 的质与量也有很大的差异。有些架构，例如 PPC 和 x86，为一系列硬件建立了各种 bootloader，有些架构仅拥有少数的 bootloader 或者没有标准的 bootloader，并且主要依靠的是硬件制造商提供的 bootloader。如果使用的是制造商提供的 bootloader，确定你得到了所有的二进制文件。如果可能的话，最好同时获得源码，以便为目标板定制 bootloader。

本章将会专注在嵌入式系统中最常用来加载 Linux 的 bootloader/boot（引导加载程序/引导）设置组合。举例来说，尽管可以在硬盘上安装和使用 GRUB，不过在嵌入式 Linux 系统中它最常被用来从 DOC 设备加载 Linux，因此讨论 GRUB 的那一节将只会包含使用 GRUB 从 DOC 设备加载 Linux 的内容。

本章一开始会检查可供 Linux 使用的大量嵌入式 bootloader。接着我们会探讨如何设置及设定一个可以提供 BOOTP/DHCP 和 NFS 服务的服务器，让目标板能够使用这些服务来获得内核映像以及安装其根文件系统。然后我们会深入探讨 LILO 与磁盘设备的使用、GRUB 与 DOC 设备的使用，以及 U-Boot 的使用。

本章结束之后，我们或者是已经安装了前面建立的所有组件、使用适当的 bootloader 设定目标板以及准备启动系统，或者是已经知道从何处获得达成此目的其余信息。

各式各样的 bootloader

正如前文所述，Linux 有许多 bootloader 可用。本节，我将会针对每个架构介绍用得最多的开放源码 bootloader。有些架构，例如 MIPS 或 m68k 处理器，根本不存在标准的 bootloader。如果目标板使用的是 MIPS 或 m68k 处理器，那么你可以参考制造商提供的文件中关于如何设置以及启动硬件的说明。

此外，有些出版物会特别区分引导加载程序和监控程序的差异：引导加载程序只是用来启动设备以及执行主要软件的组件，监控程序除了引导功能还提供用来调试、读写内存、重新编程 flash 设备、配置等等的命令行接口。这两种软件在本章都会被称这种引导加载程序，不过只要引导加载程序具备监控程序的能力，我仍会加以指明。

与引导加载程序相比，请记住在系统开发期间监控程序的能力是一笔“资产”。然而开发完成之后这些能力却成了“负担”，因为此刻的重点变成了避免用户误入监控程序模式。有些引导加载程序，例如 U-Boot，可以通过配置来启用或禁用引导加载程序特性。你或许还可以将产品制作成避免物理访问串行端口的硬件。

表 9-1 列示了 Linux 可以使用的开放源码 bootloader 以及其所支持的架构。该表还会针对每个 bootloader 指出该 bootloader 是否提供监控程序的能力并提供简短的说明。当你要决定哪个 bootloader 最适合嵌入式系统时，表 9-1 可作为一个起点。

表 9-1: Linux 的开放源码 bootloader 以及其所支持的架构

bootloader	监控程序	说明	架构			
			x86	ARM	PowerPC	MIPS SuperH m68k
LILO	否	Linux 主要的磁盘引导加载程序	×			
GRUB	否	LILO 的 GNU 版后继者	×			
ROLO	否	不需要 BIOS 可直接从 ROM 加载 Linux	×			

表 9-1: Linux 的开放源码 bootloader 以及其所支持的架构 (续)

bootloader	监控程序	说明	架构						
			x86	ARM	PowerPC	MIPS	SuperH	m68k	
Loadlin	否	从 DOS 加载 Linux	×						
Etherboot	否	经由 Ethernet 卡启动系统的 ROMable loader	×						
LinuxBIOS	否	以 Linux 为基础的 BIOS 替代品	×						
Compaq 的 bootldr	是	主要用于 Compaq iPAQ 的多功能加载程序		×					
blob	否	来自 LART 硬件计划的加载程序		×					
PMON	是	Agenda VR3 中所使用的加载程序				×			
sh-boot	否	LinuxSH 计划的主要加载程序					×		
U-Boot	是	以 PPCBoot 和 ARM-Boot 为基础的通用加载程序	×	×	×				
RedBoot	是	以 eCos 为基础的加载程序	×	×	×	×	×	×	

除了上表所列表的内容，下面是针对每个架构可使用的 bootloader 所进行的若干观察结果：

x86

x86 有两个主要的 bootloader 可用：LILO 和 GRUB。LILO 可以说是大多数 x86 工作站和服务器的发行套件上的主流 bootloader。然而在 Red Hat 的发行套件上，LILO 已经被 GRUB 取代。在某些情况下，可能会想要使用较不为人知的 bootloader，例如 Rolo 和 EtherBoot。

正如你所见，目前只有几个 x86 bootloader 提供监控程序的能力。x86 bootloader 最显著的限制就是开发时多半需要 x86 架构的主机。举例来说，LILO 和 GRUB 的 Makefile 不会被建立成允许交叉编译。此外你很难从非 x86 的主机将 LILO 或 GRUB 安装到供 x86 目标板使用的存储媒体。因此即使你所有的开发工作都是在非 x86 的主机上进行的，仍可能需要使用 x86 的主机来编译和安装你所选择的 x86 bootloader。

ARM

虽然 U-Boot 的主要目标是成为标准的 ARM bootloader，不过在撰写本书时，以 ARM 为基础的系统还没有标准的 bootloader。尽管如此，如表 9-1 所示，可以找到一对支持不同硬件的 ARMbootloader。事实上，还有许多其他的 bootloader 可以用来在 ARM 系统上启动 Linux。这些 bootloader 中有的已经老旧或很久没更新，有的只能用在特殊的电路板上或并非使用开放源码的许可条款。

PowerPC

U-Boot（之前的 PPCBoot）是大多数 PPC 系统的主要 bootloader。

MIPS

以 MIPS 为基础的嵌入式 Linux 系统还没有标准的 bootloader。尽管 PMON 可作为最初的程序代码库，不过在可以使用之前你或许需要将它移植到目标板。写作本书时，让 U-Boot 支持 MIPS 的工作正在进行中。

SupperH

尽管 sh-boot 是以 SH 为基础的嵌入式 Linux 系统的主要 bootloader，或许会发现其他的 bootloader（例如 RedBoot）更适合在系统上使用。

M68k

尽管 RedBoot 支持某些以 m68k 为基础的系统，但是以 m68k 为基础的嵌入式 Linux 系统并没有标准的 bootloader。

介绍过 bootloader 以及描述过 bootloader 所支持的架构后，接着让我们进一步检查每个 bootloader。

LILO

LinuxLOader (LILO) 由 Werner Almesberger 在 Linux 草创时期提出（译注 1）。而今 LILO 由 John Coffman 维护并可以从 <http://lilo.go.dyndns.org/> 获得最新的版本。LILO 是个拥有详细文档的 bootloader。例如 LILO 套件便附带有 user manual（使用手册）和 internals manual（参考手册）。此外，还可以在 LDP 的“LILO mini-HOWTO”以及《Running Linux》的第五章找到 LILO 的使用指南。

GRUB

GRand Unified Bootloader (GRUB) 是 GNU 计划的主要 bootloader。GRUB 最初是由 Erich Boleyn 为 GNU Mach 撰写的 bootloader。后来由 Gordon Matzigkeit 和 Okuji

译注 1：参见 <http://www.almesberger.net/cv/projects.html>。

Yoshinori 接替 Erich 的工作继续维护和开发 GRUB。你可以在 GRUB 计划位于 <http://www.gnu.org/software/grub/> 的网站找到对套件的使用有广泛探讨的《GRUB manual》说明文件。开发期间你可能会发现 GRUB 有一个功能很有用，那就是它能够使用 TFTP 和 BOOTP 或 DHCP 通过网络来引导。你可以使用 CVS 取回 GRUB 的程序代码，或是通过该计划的网站下载最新的稳定版套件。

ROLO

ROmable LOader (ROLO) 由 Sysgo Gmbh 的 Robert Kaiser 撰写和维护，它是 Sysgo 的 Elinos 发行套件的一部分。ROLO 能够不需要任何 BIOS 直接从 ROM 来启动 Linux。ROLO 可以从 <ftp://ftp.elinos.com/pub/elinos/rolo/> 获得。尽管此套件包含的文档极少，但 Vipin Malik 写了一篇文章详细说明如何在嵌入式系统中使用 ROLO（参见 http://www.embeddedlinuxworks.com/articles/rolo_guide.html）。

loadlin

loadlin 是由 Hans Lermen 在 <http://elserv.fhm.fgan.de/~lermen/> 维护的 DOS 工具程序，可用来加载 Linux。尽管你应该避免将系统建立成必须先加载 DOS，不过有些时候这样的工具程序可能会很有用。例如你想要使用 M-Systems 的 DOS 工具，以便从 DOC 设备来引导。在这样的情况下，可以修改 *autoexec.bat* 文件，让它使用 *loadlin* 工具程序来加载 Linux。然而正如我们稍后所见，可以使用 GRUB 直接从 DOC 设备启动 Linux。

EtherBoot

许多网卡上附带有可插入 ROM 芯片的插座。这些 ROM 芯片可视为 BIOS 的扩展，它们会在系统启动期间执行。EtherBoot 便是借助这种功能来支持从网络启动无盘系统。EtherBoot 已经用在了许多环境之中，包括 X 终端、路由器以及群集。完整的文件可以从 <http://etherboot.sourceforge.net/> 获得。想知道有哪些制造商销售内附 EtherBoot 的 EPROM，可参考该网站所提供的链接。

LinuxBIOS

LinuxBIOS 是完整的 BIOS 替代品，它可以在系统启动期间从 ROM 启动 Linux。LinuxBIOS 的开发是美国洛斯亚拉莫斯国家实验室的群集研究的一部分，并且受到许多硬件制造商的支持。LinuxBIOS 套件和文件可以从 <http://www.linuxbios.org/> 获得。

Compaq 的 bootldr

虽然 Compaq 的 bootldr 最初只是针对 Compaq iPAQ 进行开发、但是它目前支持 Intel 的 Assabet 以及 HP 的 Jornada 720。尽管 bootldr 支持的硬件有限，不过它却提供了非常丰富的命令集并且具备直接从 JFFS2 MTD 分区加载内核的能力。bootldr 是 <http://www.handhelds.org/> 收集软件的一部分（参见 <http://www.handhelds.org/downloads.html>），可以从 <ftp://ftp.handhelds.org/bootldr/> 下载该套件。

blob

blob 是 LART 硬件计划采用的 bootloader（注 1）。因为它的采用，blob 被移植到了许多其他使用 ARM 的系统，包括 Intel 的 Assabet 以及 Brutus、Shannon 和 Nesa。与 ARMBoot 和 Compaq 的 bootldr 不同的是，blob 并不提供监控程序的能力，尽管它可以重新编程 flash、并且能够直接从 JFFS2 MTD 加载内核。blob 及其文件可以从 LART 位于 <http://www.lart.tudelft.nl/lartware/blob/> 的网站获得。

PMON

Prom Monitor (PMON) 是 Phil Bunce 为了支持 LSI LOGIC 的 MIPS 电路板撰写的 bootloader。PMON 的发行采用了非常单纯的许可声明，它只提到 PMON 并不提供任何担保，以及你可以自由散布 PMON 不受任何限制。尽管 1999 年之后 Phil 的 PMON 已不再更新，不过你仍可以到 <http://www.carmel.com/pmon/> 获得此套件。不过有其他人在最近的计划中使用 PMON。例如它被 Bradely LaRonde 移植到了 Agenda 的 VR3 Linux PDA。这个版本的 PMON 可以从 AGOS SourceForge 位于 <http://agos.sourceforge.net/> 的工作空间获得，还可以到位于 <http://agendawiki.com/> 的 Agenda Wiki 网站找到它的使用说明。目前 PMON 仍不存在中央的权责单位或网站，而且只支持少数的电路板。正如前文所述，PMON 的程序代码库可以作为一个很好的起点，不过多半需要先将它移植到系统上，才有办法使用它。

sh-Boot

sh-boot 是 SourceForge 上 Linux SH 计划的一部分。可惜 sh-boot 已经有一段时间没更新了，所以你可能需要评估它能否用于系统。此外，sh-boot 是一个简单的 bootloader，而且它并不提供任何监控程序的能力。你可以使用 CVS 从 Linux SH 计划位于 <http://linuxsh.sourceforge.net/> 的库中获得 sh-boot 的源码。

注 1： 参见附录二对 LART 所做的说明。

U-Boot

尽管有许多其他的 bootloader，不过 Das U-Boot-universal bootloader 却被认为是功能最多、最具弹性以及开发最积极的开放源码 bootloader。它目前由 DENX Software Engineering 的 Wolfgang Denk 维护，并且受到各种开发者的支持。U-Boot 以 PPCBoot 和 ARMBoot 计划为基础。PPCBoot 本身基于 8xxrom 的源码，ARMBoot 则是 Sysgo Gmbh 将 PPCBoot 移植到 ARM 的成果。写作本书时，U-Boot 大约支持 100 种基于 PPC 的电路板、一打以上基于 ARM 的电路板以及若干基于 x86 的电路板。由于 U-Boot 可以广泛地应用在各种硬件之上，这鼓舞了开发者继续将它移植到更新的电路板和架构上。

除了别的能力之外，U-Boot 具备通过 TFTP、从 IDE 或 SCSI 磁盘以及从 DOC 启动的能力。此外它还提供 JFFS2 的只读支持。除了具备广泛的命令集以及许多能力，它也拥有相当完善的文档。附带在套件中的 *README* 提供有如何使用 U-Boot 的详细说明。套件源码树中的 *doc* 目录包含有某些电路板所需要的额外指示。除了套件提供的指示，还可以在 <http://www.denx.de/re/DPLG.html> 上看到 Wolfgang 所写的《DENX PPCBoot and Linux Guide》。该文件对如何在 TQM8xxL 电路板上使用 PPCBoot 来启动 Linux 举了许多实例。尽管该文件会假定你使用的是 PPCBoot 以及 DENX 的 Embedded Linux Development Kit (ELDK) 发行套件（注 2），不过其中与 PPCBoot 的使用有关的部分，不需要或仅需要稍做修改就可以应用在 U-Boot 之上，所以不管使用何种发行套件对你都会有所帮助。

U-Boot 套件可以从该计划位于 <http://sourceforge.net/projects/u-boot> 的工作空间获得。如果想要使用 U-Boot，将会发现订阅该网站上讨论得非常热烈的 U-Boot users 邮件论坛对你很有帮助。尽管写作本书时该网站并没有为 U-Boot 提供文件，不过你仍然可以参考两个以 U-Boot 为基础的计划，PPCBoot (<http://ppcboot.sourceforge.net/>) 和 ARMBoot (<http://armboot.sourceforge.net/>)，所提供的文件和背景知识。本章稍后将会探索 U-Boot 的使用。

RedBoot

RedBoot 应该是 Red Hat 的下一代 bootloader，目的在于取代使用固件并支持广泛硬件的 CygMon 和 GDB stub。尽管 Red Hat 已经停止了 eCos 的开发活动，不过 RedBoot 却是以 eCos 为基础，现在 eCos 的许可条款已经改采 GPL 的方式，并且交由 Red Hat 的 core eCos 开发人员继续维护。因此 eCos 及 RedBoot 的未来，维系在这些开发人员手上。

注 2: ELDK 是个开放源码的开发及目标板发行套件。

尽管与 eCos 有依存关系（注 3），RedBoot 仍不失为一个非常有用的 bootloader。举例来说，各种开放源码 bootloader 中，目前只有 RedBoot 支持第三章所提到的各种架构，以及基于这些架构的各种电路板。此外 RedBoot 套件拥有相当完善的文档，如《RedBoot User's Guide》便针对了一打以上的不同系统提供它的使用实例。RedBoot 的网站位于 <http://sources.redhat.com/redboot/>，可以使用 CVS 取回它的源码，它的源码就包含在 eCos 的源码中。最近 Red Hat 的 core eCos 开发人员成立的 eCosCentric Ltd. 公司已经在 <http://www.ecoscentric.com/snapshots/> 提供 CVS snapshot（当日最新版）。

网络引导的服务器设置

正如我们在第二章所见，开发初期适合将目标板设置成网络引导，因为这样你可以逐步修改内核以及根文件系统，不必在每次修改之后更新目标板的存储设备。尽管并非所有的 bootloader 都可以使用这种方式引导，不过建议尽可能采用这种设置方式。

正如我前面所说，从网络启动目标板的最简单方法就是使用 BOOTP/DHCP、TFTP 和 NFS。其中，BOOTP/DHCP 协议是用来为网络主机提供基本引导信息的标准方法，包括其他服务器（例如 TFTP 和 NFS）的地址。TFTP 是用来下载远程文件的最简单网络协议。对嵌入式 Linux 系统来说，目标板可通过 TFTP 协议从 TFTP 服务器获得内核映像。NFS 则是客户端和服务端之间用来共享整个目录树的标准和最简单的协议。对嵌入式 Linux 系统来说，目标板可通过 NFS 协议来安装 NFS 服务器为它导出的根文件系统。然而 NFS 只能在一个已经启动的 Linux 内核上运行。并用这三个协议可以提供非常有效的主机/目标板开发设置。

要启用目标板的网络引导功能，必须为开发主机设置相应的网络服务，这样目标板才能存取到它需要的组件。尤其是，需要将主机设置成响应 BOOTP/DHCP 的请求、使用 TFTP 服务器提供内核，以及启用 NFS 安装功能。下面将会分节探讨这些问题。

设置 DHCP 监控程序

与其他的网络服务不同，DHCP 与 internet super-server 之间并无依存关系。要提供 DHCP 服务，必须手动执行 DHCP 监控程序。首先确定系统上是否安装了 DHCP 服务器。尽管你可以从 <http://www.isc.org/> 下载 DHCP 套件，不过大多数主流发行套件通常会附带 DHCP 服务器。

注 3： RedBoot 系 eCos 源码树的一部分，因为它需要 eCos 提供部分的程序代码。因此 RedBoot 的开发受限于但并不完全依赖 eCos 的开发。例如，RedBoot 所支持的某些平台，eCos 并不支持。

如过你使用的是基于RPM的发行套件，可以使用如下的命令检查DHCP监控程序是否存在：

```
$ rpm -q dhcp
dhcp-2.0-5
```

此例中，系统已经安装DHCP 2.0-5。如果系统尚未安装DHCP，请针对发行套件使用适当的工具来安装DHCP服务器。请注意，大多数的发行套件都会附带两种DHCP套件：客户端和服务端。包含客户端的套件通常称为*dhcpc-VERSION*。它是以在“dhcp”之后附加“c”来代表客户端套件。

要想系统正常运行，用来执行DHCP服务器的内核必须设定CONFIG_PACKET和CONFIG_FILTER选项。大多数发行套件附带的内核几乎总是会启用这些选项。如果习惯为工作站建立你自己的内核，正如我往常所做的，建立内核的时候请密切注意这些选项。如果内核建立不当，DHCP监控程序启动时将会输出如下的信息：

```
socket: Protocol not available - make sure CONFIG_PACKET and CONFIG_FILTER
are defined in your kernel configuration!
exiting.
```

安装好套件并正确配置好内核后，接着建立或编辑*/etc/dhcpd.conf*文件，为目标板加入一项设定。例如下面是针对我的控制模块设定的*/etc/dhcpd.conf*文件：

```
subnet 192.168.172.0 netmask 255.255.255.0 {
    option routers 192.168.172.50;
    option subnet-mask 255.255.255.0;

    host ctrl-mod {
        hardware ethernet 00:D0:93:00:05:E3;
        fixed-address 192.168.172.10;
        option host-name "ctrl-mod";
        next-server 192.168.172.50;
        filename "/home/karim/vmlinux-2.4.18.img";
        option root-path "/home/karim/ctrl-rootfs";
    }
}
```

基本上，这项设置指出主机和目标板位于192.168.172.0网络上，TFTP服务器位于192.168.172.50，当目标板送出DHCP或BOOTP服务请求时会被配置成192.168.172.10这个地址。其中，hardware ethernet字段的设定值（即MAC地址）用来标识目标板，00:D0:93:00:05:E3是我的控制模块的MAC地址。fixed-address字段用来告诉DHCP服务器应该将哪个IP地址分配给所指定的MAC地址。option host-name字段用来为目标板指定供*/etc/dhcpd.conf*文件使用的主机名称。next-sever字段用来告诉

目标板TFTP服务器的地址。`filename`字段用来指定目标板下载的内核的文件名(注4)。DHCP的规范RFC 2131指出`filename`字段值的长度不能超过128个字节。最后,`option root-path`字段用来提供NFS服务器上目标板根文件系统的完整路径。如果目标板不需要从NFS服务器下载它的根文件系统,可以省略最后一个字段。因为此例中主机与目标板只有一条网络连接,所以`option routers`字段会指向主机的IP地址。如果目标板实际上是通过路由器连上整个网络的,则`option routers`字段应该指向该网络的预设路由器。

上面提供的范例配置应该轻易就能够在你自己的目标板上使用。如果需要有关DHCP服务器配置的更多信息,请参考`dhcpd.conf`的manpage以及发行套件安装的范例配置文件(如果有的话)。

请注意,如果使用的DHCP监控程序是3.0b2pl11之后的版本,例如Red Hat 8.0附带的版本,将需要将下面这行设置加入`dhcpd.conf`文件中:

```
ddns-update-style ad-hoc;
```

为目标板设好DHCP服务器之后,几乎就已经做好了启动DHCP服务器的准备工作。然而在启动DHCP服务器之前,必须已经存在`/var/state/dhcp/dhcpd.leases`文件。如果不存在,可以使用`touch`命令来产生它。如果并未产生该文件,DHCP监控程序将会拒绝启动。

最后启动DHCP服务器。对以Red Hat为基础的发行套件来说,请键入:

```
# /etc/init.d/dhcpd start
```

设置 TFTP 监控程序

设置TFTP监控程序的第一步就是确定TFTP套件已经安装。尽管最新版的TFTP监控程序可以从<ftp://ftp.uk.linux.org/pub/linux/Networking/netkit/>以下载NetKit套件的方式获得,不过TFTP很有可能已经随发行套件一起安装在系统上,或是你可以使用发行套件的光盘来安装它。

如果使用的是以RPM为基础的发行套件,请使用如下的命令来检查TFTP监控程序是否存在:

```
$ rpm -q tftp
tftp-0.16-5
```

注4: 因为版面的宽度有限,所以我并没有使用完整的`/home/karim/control-project/control-module/...`路径。实际使用时,请根据自己的情况使用完整的路径。

此例中、系统已经安装 TFTP 0.16-5。如果系统尚未安装 TFTP，请针对发行套件使用适当的工具来安装 TFTP 套件。如果系统并不依赖套件管理程序、或者系统上某些组件并非通过套件管理程序安装的，还可以使用 *whereis* 命令来检查系统上是否存在 TFTP 监控程序。

安装好套件之后，可以通过修改适当的 *internet super-server* 配置文件来启用 TFTP 服务。简单地说，*internet super-server* 的功能就是代替网络服务监听其提供服务的端口号。当某个服务的请求被收到时，*super-server* 会启动相应的监控程序来处理此项请求。因此任何时间执行的监控程序数量都是最少的。TFTP 服务按惯例交由 *super-server* 代为处理。

要在使用 *inetd* 的系统中启用 TFTP 服务，请编辑 */etc/inetd.conf*，移除 TFTP 服务那一行的注释符号（也就是拿掉该行开头处的 # 符号），并且送出 *SIGHUP* 信号给 *inetd* 进程，要求它重新读取配置文件。要在使用 *xinetd* 的系统中启用 TFTP 服务，请编辑 */etc/xinetd.d/tftp*，将包含 *disable = yes* 的那一行注释掉（也就是在该行开头处附加 # 符号）。如同 *inetd*，必须送出 *SIGHUP* 信号给 *xinetd*。

最后你必须将一份目录清单（这些目录包含可供 TFTP 客户端下载的文件）提供给 TFTP 服务器。在使用 *inetd* 的系统中，请将目录清单附加在 */etc/inetd.conf* 文件中 TFTP 服务那一行。在使用 *xinetd* 的系统中，请编辑 */etc/xinetd.d/tftp* 文件，将目录清单附加在 *server_args =* 那一行。TFTP 的缺省目录是 */tftpboot*。你可能需要根据设置修改这项设定。不论你选用哪个目录，确定它的存取权限包括“other”的读写权限。

在使用 *inetd* 的主机上，可以在 */etc/inetd.conf* 文件中看到如下的 TFTP 设定：

```
tftp    dgram  udp      wait    root    /usr/sbin/tcpd  in.tftpd /home/karim/
```

此例中，内核映像在 */home/karim* 目录中，它具有如下的存取权限：

```
$ ls -ld /home/karim
drwxr-xr-x    4 karim    karim          4096 Aug 29 16:13 karim
```

在使用 *xinetd* 的主机上，下面是经过修改的 */etc/xinetd.d/tftp*（安装自以 Red Hat 为基础的套件）：

```
service tftp
{
    socket_type        = dgram
    protocol           = udp
    wait               = yes
    user               = root
    server              = /usr/sbin/in.tftpd
    server_args        = /home/karim
    # disable          = yes
    per_source         = 11
```

```
        cps                = 100 2  
    }  
}
```

不管主机使用何种super-server, 缺省情况下TFTP服务通常是禁用的。因此即使你使用缺省的 */tftpboot* 目录, 仍需要修改super-server的配置文件以启用TFTP。

安装 NFS 服务器上的根文件系统

正如我在第二章的说明, 尽管bootloader与内核必须通过前面提到的某个方法存入或取自当地的存储设备, 不过目标板的内核却可以从远程NFS服务器安装它的根文件系统。为此目的, NFS服务器必须有正确的安装和配置设定。第六章有提到如何为目标板建立根文件系统。尽管第八章提到如何针对目标板的使用来准备根文件系统, 不过我们在第六章建立的根文件系统并不需要经过任何特别的准备就可以供NFS服务器使用。

NFS服务器有两种用法: 成为独立的用户应用程序, 或者成为内核的一部分。除了速度比较快, 后者也是大多数发行套件的标准用法。除了NFS服务器本身, 还需要安装NFS工具程序。通常发行套件会附带内含NFS工具程序的nfs-utils套件。请使用如下的命令来检查nfs-utils是否存在:

```
$ rpm -q nfs-utils  
nfs-utils-0.3.1-13
```

确定系统安装了nfs-utils后, 还得确定是否存在适当的配置文件以及是否启动了相应的服务。

*/etc/exports*是我们需要为NFS服务器设定的主要文件。你可以在此文件的设定项中描述每个主机或是一群主机可以存取的目录。例如, 下面是我为控制模块在*/etc/exports*文件中所加入的一项设置:

```
/home/karim/ctrl-rootfs 192.168.172.10(rw,no_root_squash)
```

这项设置指出: 位于192.168.172.10的主机对*/home/karim/ctrl-rootfs*目录(指向我们在第六章为目标板所建立的根文件系统)具有读写存取权限。另外, *no_root_squash*参数用来要求服务器允许远程系统以它自己的root特权存取该目录。请注意, 此处授予了目标板非常大的权利。如果我们全盘控制设备的存取管道, 正如大多数的开发设置, 这么做显然不会有任何危险。然而, 如果目标板摆在比较不安全的地方, 或是它被直接连上Internet, 最好使用缺省的*root_squash*测试。这样的话, 目标板自己的根文件系统可能有一大半都无法写入, 不过它仍然可以读写可供任何人读写的目录和文件。然而, 从实际上来看, 目标板的运行将会受到极大的限制。

因为提供NFS服务还会涉及网络滥用的危险, 所以使用若干最起码的保护机制来避免侵入通常是比较恰当的做法。完成此事的一个方法就是设定*/etc/hosts.deny*和*/etc/*

hosts.allow 文件来限制网络服务的存取权限。例如下面是我的 Red Hat-based 主机的 */etc/hosts.deny* 文件:

```
#
# hosts.deny
#

portmap: ALL
lockd: ALL
mountd: ALL
rquotad: ALL
statd: ALL
```

下面是我的 */etc/hosts.allow* 文件:

```
#
# hosts.allow
#

portmap: 192.168.172.10
lockd: 192.168.172.10
mountd: 192.168.172.10
rquotad: 192.168.172.10
statd: 192.168.172.10
```

这两个文件指定的规则限制了各种文件共享服务的存取权限。一起使用这两个文件会产生如下的结果: 只有位于 192.168.172.10 的机器可以使用 NFS 服务。就我的设置来说这么做很好, 因为我不想和任何其他人共享我的工作站。就算你不想设定 */etc/hosts.deny* 和 */etc/hosts.allow*, 还是建议你将安全问题放在心上以及使用任何必要的方法, 例如进行备份, 来保护成果。

建立好配置文件之后, 接着启动 portmapper 服务 (这是 NFS 服务器本身需要的服务):

```
# /etc/init.d/portmap start
```

最后启动 NFS 服务器:

```
# /etc/init.d/nfs start
```

如果想要进一步了解使用 NFS 进行远程引导的配置, 可参考 LDP 的两篇 “Diskless root NFS HOWTO” 文件。此外也可参考 LDP 的《NFS HOWTO》文件。

在磁盘和 CompactFlash 设备上使用 LILO

因为已经有大量的文件在说明 LILO 的安装、配置和使用, 所以我将只会说明 LILO 在嵌入式 PC-like 系统中的特殊用法。尤其是, 我将会告诉你如何在主机上把 LILO 安装到目标板将会使用的存储设备。

将 LILO 安装在目标板的存储设备上，需要使用第二章提到的可抽换存储设备设置。在这种设置中，目标板的存储设备可从目标板抽出并且连接至主机以便进行编程。因此，目标板的存储设备会受到主机操作系统的控制，就像任何其他的主机设备一样。目标板的存储设备因此会被主机视为额外的存储设备。例如它可以被当作第二台 IDE 磁盘 (*/dev/hdb*) 或是第一台 SCSI 磁盘 (*/dev/sda*)。不管它被主机的内核当作什么，LILO 要将自己安装到辅助存储设备（而非主机缺省的引导媒体）上将需要使用特殊的方法。

正如第八章所说，CF 设备就这一点来说相当奇怪，因为当通过 USB CF 读卡器存取时，它们在主机上会被当作 SCSI 磁盘 (*/dev/sdX*)，然而当通过 CF-to-IDE 或 CF-to-PCMCIA 适配卡存取时，它们在目标板上却被当作 IDE 磁盘 (*/dev/hdX*)。我在下面提供的配置文件范例可以解决这个问题，通过适当的 BIOS 和内核标志让 CF 设备可以在主机上被视为 SCSI 磁盘，一旦放回目标板又可以当作 IDE 磁盘正常引导。

接下来，我会假定主机可以存取将会安装 LILO 的存储设备，而且你使用的 LILO 是 22.3 或之后的版本。如果是之前的版本，有一个重要的命令将会执行失败，稍后会有简短的说明。下面说明如何在主机上将 LILO 安装到第二台 IDE 或 SCSI 存储设备：

1. 为安装 LILO 的存储设备在目标板的根文件系统中建立适当的 */dev* 条目。这个存储设备并非你在目标板中存取的那个存储设备。更确切地说，主机会使用这个存储设备条目来存取特定的存储设备。举例来说，如果想将 LILO 安装到 */dev/sda*（通常就是你系统中的第一台 SCSI 硬盘），目标板的根文件系统上必须存在 */dev/sda* 条目。这个条目在目标板上很可能没有相应的设备可用。的确，有可能在主机上它被当作 */dev/sda* 来用，在目标板上它被当作 */dev/hda* 来用。尽管如此，必须在目标板的根文件系统中建立 */dev/sda* 条目，这样才能在主机上安装 LILO。要进一步了解 */dev* 条目与物理存储设备的关系可参考《Running Linux》第三章。
2. 在目标板的根文件系统上建立 LILO 配置文件。当你将 LILO 安装到目标板的存储设备时，要避免覆盖主机的配置，请将 LILO 配置放到目标板根文件系统中的 */etc/target.lilo.conf* 文件里，不要放到 */etc/lilo.conf* 中。因此，如果不小心执行一条修改主机配置的 LILO 命令，它将会报告找不到文件，主机也不会受到危害。

下面这个 */etc/target.lilo.conf* 配置文件范例会从 CF 卡启动我的 DAQ 模块：

```
boot = /dev/sda
disk = /dev/sda
    bios = 0x80

image = /boot/bzImage-2.4.18
    root = /dev/sda1
    append = "root=/dev/hda1"
    label = Linux
    read-only
```

此例中，通过 USB CF 读卡器存取 CF 卡，在我的主机上 CF 卡会被视为 SCSI 磁盘 `/dev/sda`。然而在目标板上，是通过 CF-to-IDE 适配卡存取 CF 卡的，而且 CF 卡会被视为 IDE 磁盘 `/dev/hda`。如果使用标准的 LILO 配置文件来设定 LILO，它将会猜测其所在磁盘的 BIOS ID，并且在启动时使用该 ID（识别码）给 BIOS 送出存取请求。因为，此例中，它运行在 SCSI 磁盘上，它将会假设一个 SCSI BIOS ID，并针对此类磁盘送出存取请求。因为目标板上并不存在此类磁盘，所以 BIOS 会传回错误信息，造成 LILO 引导失败。诀窍就是在配置文件中加入一行 `bios = 0x80` 设定。这样 LILO 就会从 BIOS ID 为 0x80 的磁盘（也就是系统中第一台 IDE 磁盘）引导。因为此处会有 SCSI 和 IDE 混淆的情况，所以我还必须为内核的引导参数附加 `root=/dev/hda1` 选项。否则内核将无法找到它的根文件系统，并且会在尝试安装它的时候崩溃（注 5）。

另外，如果要将 LILO 安装到 `/dev/hdb`，请将前面的 `/dev/sda` 设定项替换成 `/dev/hdb`。这样的话，将不需要为内核的引导参数附加 `root=/dev/hda1` 选项，因为该磁盘在主机和目标板上都会被视为 IDE 设备。

当 LILO 以上面的配置文件执行时，它会启动主机的 `/dev/sda` 设备，并且在该处安装它自己。因为这个配置文件是 `${PRJROOT}/rootfs/etc/target.lilo.conf` 而不是 `/etc/lilo.conf`，所以必须使用特殊的选项为 LILO 指定配置文件的位置。稍后再示范使用这个配置的完整 LILO 命令行。

欲了解如何将 LILO 安装到其他存储设备，可参考 LDP 提供的“LILO mini-HOWTO”文件的“Installing hdc to Boot as hda and Using bios=”这一节。

3. 如果有必要的话，使用 `fdisk` 对存储设备进行分区。
4. 使用适当的文件系统建立工具为你所选用的文件系统类型在存储设备上建立文件系统。例如，若选用 ext2 文件系统，则使用 `mke2fs`。
5. 将根文件系统分区安装到 `/mnt` 中适当的目录上。
6. 使用 `cp -a` 将根文件系统复制到你指定的分区。根文件系统必须包含前面建立的 `/etc/target.lilo.conf` 文件所提及的内核映像，此例为 `/boot/bzImage-2.4.18`。
7. 将 LILO 安装到存储设备。以我的 DAQ 模块的存储设备为例，在我的主机上它会被安装成 `/mnt/cf`，而且我会使用如下的命令：

```
# lilo -r /mnt/cf -C etc/target.lilo.conf
Warning: etc/target.lilo.conf should be owned by root
```

注 5：一般情况下，如果在映像的描述中已经存在一行 `root` 的设定，应该不需要再为内核的开机参数附加 `root=` 选项。然而，此例中，相关软件会假定磁盘无法变更类型，如果不使用双重声明将无法正确设定开机过程。

```
Warning: LBA32 addressing assumed
Added Linux *
```

这个命令会要求 *lilo* 使用 `chroot()` 系统调用将它的根目录改为 */mnt/cf* 目录，并且使用该目录中找到的 *etc/target.lilo.conf* 配置文件。这个命令会编程 *target.lilo.conf* 配置文件中指定的设备。配置文件中指定的 */dev* 条目是以根目录 */mnt/cf* 为起点。举例来说，如果必须编程 */dev/sda*，LILO 会启动并编程 */mnt/cf/dev/sda*。

如果忘了在目标板的根文件系统上建立 *target.lilo.conf* 所指定的 */dev* 条目，这条命令将会执行失败。如果目标板的根文件系统上没有 */tmp* 目录，这条命令也会执行失败。此外，如果使用的 LILO 是 22.3 以前的版本，这条命令将会报告如下的错误信息并且会执行失败：

```
Fatal: open /boot/boot.b: No such file or directory
```

这条错误信息起因于一个事实：22.3 以前的版本，LILO 的组件会分文件存放，其中包含 *.b* 文件。22.3 以后的版本，所有的 *.b* 文件变成了 *lilo* 二进制文件的一部分。

8. 卸载根文件系统分区。

你现在可以从主机取出存储设备（将主机关机并取出硬盘或是从 CF 读卡器取出 CF 卡），接着将它安装到目标板，然后开机。

在 DiskOnChip 设备上使用 LILO

要从 DOC 设备引导，必须修补 LILO（因为它并不支持 DOC）。这包括 M-Systems 提供的 Linux 工具套件，以及 MTD 套件为 LILO 提供的补丁。大家在 MTD 邮件论坛的共同经验是，GRUB 是最受 MTD 开发团队注意的 bootloader。因此我强烈建议在 DOC 设备上使用 GRUB（不要使用 LILO）引导。如果仍旧想要使用 LILO，则可以在 MTD 邮件论坛以往的记录中找寻相关的信息；如果邮件论坛记录中找不到你需要的信息，则可以向邮件论坛提出问题请求指导。

在 DiskOnChip 设备上使用 GRUB

因为 GRUB 在传统磁盘设备上的使用，GRUB 手册已经有完整的说明，所以我们将只会专注于 GRUB 在 DOC 设备上的安装和使用。在我开始说明如何在 DOC 设备上编译和使用 GRUB 的细节之前，我必须警告你，如果 GRUB 配置不当，将会导致系统无法引导。现在让我们来检查为什么会这样以及如何避免此问题。

第七章在描述 *doc_loadbio* 命令的使用时提到，DOC 设备中包含了一个称为 IPL 的 ROM 程序，系统启动时它会被视为 BIOS 的扩展，由 BIOS 负责执行。当它执行时，IPL 会安

装另一个程序SPL。要在DOC设备上使用GRUB引导，SPL必须被替换成“为了从DOC引导而定制的”GRUB版本。

因为系统中可能还会有其他的BIOS扩展，IPL加载的SPL可能无法立即启动系统，因此它必须安装 Terminate and Stay Resident（终止并驻留，TSR）程序并且进入休眠状态直到BIOS可以启动系统为止。以GRUB为例，GRUB SPL会被替换成BIOS的引导中断INT 19h，使用定制的中断处理例程来执行GRUB，以便完成从DOC设备引导的工作。因此其他的BIOS扩展能够执行，而且只有在系统可以引导时才会调用GRUB。

然而这个方法的问题是，BIOS安装的缺省引导处理例程根本没机会执行，而且当GRUB的处理例程被调用时，在BIOS中选用的任何引导配置选项（例如从硬盘或软盘先引导）将会被GRUB完全忽略掉。如果DOC上的配置文件设定正确，这样不会有什么问题。否则你需要先使用DOC引导，然后在Linux中修改配置文件，或是你也可以从DOC中将GRUB完全移除。

然而GRUB的配置文件中只要有错误就会造成引导失败的结果。如果没有办法在启动的时候停止引导中断处理例程的取代功能，将无法重新启动系统。已知有下面四种方法可以完成此事：

- 启动系统之前，可以先将DOC从系统移除。这个方法的问题是，如果手上没有DOC编程器，重新编程DOC的惟一方法就是在系统启动之后插入DOC。换言之，必须将DOC直接连上运行中的电路板。不用说，DOC和电路板接口的设计并没有考虑到这种操作方式。此外，我并不建议你这么做，如果非要这么做，我也不想担任何责任。然而在MTD邮件论坛上有一些英勇的人报告说，他们将DOC插入运行中的系统，以此方式完成重新编程的工作。
- 如果有跳线可用来设定DOC设备映射的寻址范围，可以试着完全移除跳线并且启动系统。这么做有时（例如当你使用的是M-Systems提供的ISA DOC评估板）会使得BIOS无法认出IPL来，因此不会执行IPL。然而这么做有时可能会造成系统崩溃。如果这个诀窍对你适用，将可以使用BIOS的配置来启动系统。然而，一旦系统运行之后想要存取DOC，又得在系统有电源供给的情况下插入跳线。同样地，尽管有人报告说这么做可行，但是硬件的设计并没有考虑到这种操作方式，因此我并不建议你这么做，也不想对由此带来的后果承担任何责任。
- GRUB的配置允许它使用ROM BASIC中断INT 18h来替代引导中断。近来，INT 18h除了作为ROM BASIC中断，有时也可用作网络引导。当你将GRUB设定成使用这个中断，只有在BIOS的配置被设定成网络引导或是BIOS中没有设定引导设备时才会启动GRUB。这个方法有一些缺点。首先，每当你想将“从DOC引导”切换成“从硬盘引导”时，就需要变更BIOS配置。在系统开发期间这么做很浪费时

间。其次，近来的 BIOS 对 INT 18h 的使用并未标准化，此例的 BIOS 使用 INT 18h 来提供网络引导。

- 以上就是写作本书时已经知道的方法，为了替读者找到一个比较完美的方法，因此我开始研读 DOS 和 BIOS 方面的技术手册，并提出了既漂亮又简单的解决方案。基本上，我不是全部取代缺省的引导中断处理例程，经我修改过的 GRUB SPL 会复制原始的处理例程，并且将它替换成 GRUB 引导处理例程，让 BIOS 在系统中继续找寻其他的 BIOS 扩展。当 GRUB 的引导处理例程被调用时，它会检查用户是否按下了 Ctrl 键。如果有，便会回到原始的引导处理例程，于是离开 BIOS 并且以用户选择的引导配置继续引导。如果没有按下 Ctrl 键，GRUB 会继续它的正常程序加载 DOC 上的内容。正如你所看到的，这个解决方案并未涉及任何危险的硬件操作；或许可以让人们免去“腕管综合症”（译注 2）之苦。

基于一些明确的理由，强烈建议读者使用最后一个解决方案。写作本书时，可以在 MTD CVS 里的 GRUB 补丁文件 *grub-2002-10-08-doc.patch* 中找到我在这方面所做的修补。我将会在下一节说明如何在 GRUB 配置设定期间启用这个选项。

说明过如何在 DOC 设备上使用 GRUB 引导之后，让我们来讨论 GRUB 在 DOC 设备上的建立、安装和使用。

为 DOC 设备设定和建立 GRUB

正如前文所述，将需要一台 x86 主机来建立 GRUB。接下来的说明会假定你使用的是一台 x86 主机。在任何其他类型的主机上，GRUB 的建立工作将会失败，或者建立出无法使用的二进制文件。

首先将 GRUB 下载至 `$(PRJROOT)/bootldr` 目录，并且在该处取出源码。然后将 GRUB 的补丁文件从 `$(PRJROOT)/sysapps/mtl/patches` 目录复制到 `$(PRJROOT)/bootldr` 中的 GRUB 源码目录里。以我的 DAQ 模块为例，我使用的是 GRUB 0.92 和 *grub-2002-02-19-doc.patch* 补丁文件。现在进行 GRUB 的修补：

```
$ cd $(PRJROOT)/bootldr/grub-0.92
$ patch -p0 < grub-2002-02-19-doc.patch
```

因为这个补丁文件原本是供 GRUB 0.90 使用的，当它应用在 0.92 版时会产生若干警告信息以及一项失败。这项失败出现在 *ChangeLog* 中，因此可以忽略不管。

如果想使用前一节所提到的 Ctrl 键方案来避免热插拔 DOC，必须使用从 CVS 库中取回

译注 2：手指头麻木酸痛。

的 *grub-2002-10-08-doc.patch* 补丁文件，或是较新版的补丁文件。然而因为 CVS 库时有变动，这个补丁文件不见得可以应用在最新版的 CVS 内容。尽可能取回能够完全应用的补丁文件，让修补过的源码树能够完成编译。例如我从 CVS 库取回 2002 年 10 月 10 日那一天的 GRUB 源码，然后手动编辑源码树中若干文件。为了取回 2002 年 10 月 10 日那一天的源码，我会使用如下的命令：

```
$ cvs -z3 -d:ext:anoncvs@savannah.gnu.org:/cvsroot/grub \  
> co -D"10/10/02" grub
```

程序代码修补过之后，接着建立 GRUB。首先使用自动建立工具产生 Makefile：

```
$ aclocal && automake && autoconf
```

现在针对 DOC 的需要来设定 GRUB：

```
$ ./configure --enable-diskonchip-2000 \  
> --enable-diskonchip-ctrlbypass \  
> --enable-ext2fs \  
> --disable-ffs --disable-xfs --disable-jfs --disable-vstafs \  
> --disable-reiserfs --disable-minix --disable-fat
```

这条命令行会停用 GRUB 对所有文件系统的支持（ext2 除外），并启用对 DOC 2000 设备的支持。它还使用 *--enable-diskonchip-ctrlbypass* 选项来启用前一节所提到的 Ctrl 键方案。另外还有几个与 DOC 有关的配置选项。举例来说，如果使用的是 DOC Millennium，可能会想使用 *--enable-diskonchip-mil256* 或 *--enable-diskonchip-mil512* 选项，这取决于 DOC Millennium 的分页大小是 256 或 512 个字节。你还可以使用 *--enable-diskonchip-biosnetboot* 选项让 GRUB 通过“网络引导中断”（不要通过“引导中断”）来引导，正如前文所述。想要知道 GRUB 的配置中有哪些选项跟 DOC 有关以及它们的完整说明请参考 *README_DiskOnChip*（这是前面应用 DOC 补丁文件之后在 GRUB 源码树中所产生的文件）。

设定好配置之后，接着建立 GRUB：

```
$ make
```

编译好之后，*stage1/grub_firmware* 文件的内容就是将被写入 DOC 的 GRUB 映像。请将这个文件复制到 *\$(PRJROOT)/images/grub_firmware-0.92* 以供日后使用：

```
$ cp stage1/grub_firmware $(PRJROOT)/images/grub_firmware-0.92
```

将 GRUB 安装到 DOC 设备

我已经在第七章中“安装 bootloader 映像”小节说明过了如何安装 GRUB bootloader 的映像。请根据那一节的说明将此处所建立 GRUB 映像安装到 DOC 设备。

将 GRUB 设定成从 DOC 引导

如同 LILO 一样，GRUB 也会使用配置文件来决定引导媒体以及它必须启动的内核。然而与 LILO 不同的是，不需要执行 GRUB 二进制文件来剖析和更新它的配置。取而代之的是，GRUB 的配置文件 *menu.lst* 会被原封不动地放到目标板根文件系统中的 */boot/grub* 目录里，而且 GRUB 会在启动时读取它。要将 GRUB 设定成从 DOC 引导，我们就必须建立这个文件。

在下面这个 *menu.lst* 范例文件中，你可以看到如何将 GRUB 设定成从 DOC 设备引导：

```
timeout 5
default 0

title DiskOnChip 2000 Boot
kernel (dc0,0)/boot/bzImage-2.4.18 root=/dev/nftl1a1

title HD Boot
kernel (hd0,0)/boot/bzImage-2.4.18 root=/dev/hda1
```

这个文件描述了两种引导方式。第一种方式，也是缺省的方式，用来从第一个 DOC 的第一个分区（dc0）启动内核 */boot/bzImage-2.4.18*。第二种方式，用来从第一个硬盘的第一个分区（hd0）启动与前一个选项同名的内核。对每种配置来说，*root=* 选项用来告知启动中的内核它的根文件系统可以在哪个设备上找到。

这个配置在开发期间很有用，因为它允许你选择从 DOC 或从硬盘引导。在产品系统上，或许会想要移除硬盘的设定条目，以及将超时设为 0，让“从 DOC 引导”变成惟一的选项。

你可以进一步变更 GRUB 的配置并允许若干引导选项。配置文件格式的完整说明请参考 GRUB 的使用手册。

U-Boot

正如前文所述，U-Boot 是个具备详细说明的 bootloader。例如该套件附带的 *README* 文件中详细说明了 U-Boot 的用法。除了别的之外，它还探讨了套件源码的布局、可用的建立选项、U-Boot 的命令集以及 U-Boot 特有的环境变量。接下来我将会提供 U-Boot 的必要信息以及它的使用实例。然而，要深入探讨 U-Boot，手边最好准备一份可以随时参考的数据。因此建议读者将 U-Boot 提供的 *README* 副本打印出来，并且参考计划维护者撰写的其他文档。

编译与安装

首先在 `$(PRJROOT)/bootldr` 目录中下载并解开 U-Boot 最新的版本。写作本书时，U-Boot 最新的版本是 0.2.0。取出源码之后，接着移往套件源码目录：

```
$ cd $(PRJROOT)/bootldr/u-boot-0.2.0
```

RAM 和 Flash 的物理地址

接下来的说明中使用的电路板具有 16 MB 的 RAM 和 8 MB 的 flash。RAM 的寻址空间为 0x00000000 到 0x00FFFFFF，flash 的寻址空间为 0x40000000 到 0x407FFFFFFF。U-Boot 提供的文档探讨了目标板上物理地址的用法。

建立 U-Boot 之前，需要针对目标板来设定 U-Boot 的配置。U-Boot 套件中有为大量的电路板提供预先设定好的配置。所以目标板可能已经存在非常好的配置。请检查 *README* 文件，看看电路板是否受到支持。U-Boot 的 Makefile 会对每个受到支持的目标板提供相应的 *BOARD_NAME_config* 建立目标（可用来为所指定的电路板设定 U-Boot 的建立方式）。例如我的控制模块使用的是 TQM860L 电路板，因此它的建立目标就是 *TQM860L_config*。决定好所要使用的建立目标之后，接着进行 U-Boot 的建立工作：

```
$ make TQM860L_config
```

现在建立 U-Boot：

```
$ make CROSS_COMPILE=powerpc-linux-
```

除了会产生 bootloader 映像，建立的过程中还会编译若干工具，让主机在二进制映像加载到 U-Boot 的目标板之前，先使用这些工具对它进行调整。表 9-2 列出了 U-Boot 编译期间产生的文件。

表 9-2: U-Boot 编译期间所产生的文件

文件名	说明
<i>System.map</i>	内核的符号表
<i>u-boot</i>	使用 ELF 二进制文件格式的 U-Boot 映像
<i>u-boot.bin</i>	U-Boot 的原始二进制映像，可写入引导存储设备
<i>u-boot.srec</i>	使用 Motorola 的 S-Record 格式的 U-Boot 映像

现在可以使用适当的程序将 U-Boot 映像加载到目标板的存储设备。如果目标板上已经安装了 U-Boot 或者它的原型 (PPCBoot 或 ARMBoot)，可以使用已安装的副本将 U-Boot

升级到新版本（参见“更新 U-Boot”一节的说明）。如果目标板上安装有另一个 bootloader，请遵照该 bootloader 文档描述的程序来更新 bootloader。最后，如果目标板尚未安装任何 bootloader，那么你需要使用硬件的编程设备，例如 flash 编程器或 BDM 调试器，将 U-Boot 复制到目标板。

不管你以何种方式将 bootloader 映像复制到目标板，请将各种相关的 bootloader 映像复制到 `${PRJROOT}/images` 目录。以我的控制模块为例，我会以如下的方式来复制相关的映像：

```
$ cp System.map ${PRJROOT}/images/u-boot.System.map-0.2.0
$ cp u-boot.bin ${PRJROOT}/images/u-boot.bin-0.2.0
$ cp u-boot.srec ${PRJROOT}/images/u-boot.srec-0.2.0
```

如果想要对 U-Boot 本身进行调试，还必须复制 ELF 二进制文件：

```
$ cp u-boot ${PRJROOT}/images/u-boot-0.2.0
```

最后是 U-Boot 建立时产生的主机工具：

```
$ cp tools/mkimage ${PREFIX}/bin
```

用 U-Boot 引导

一旦 U-Boot 被正确安装到目标板之后，接着可以使用串行线连接目标板，并且使用终端仿真程序连接目标板，然后用 U-Boot 引导。正如我在第四章所说，并非任何终端仿真程序都能够与任何 bootloader 交互。以 U-Boot 为例，应该避免使用 *minicom* 进行文件传输，因为当你进行这样的传输时可能会发生问题。

下面是我的控制模块在引导期间的输出：

```
U-Boot 0.2.0 (Jan 27 2003 - 20:20:21)

CPU:   XPC860xxZPnnD3 at 80 MHz: 4 kB I-Cache 4 kB D-Cache FEC present
Board: TQM860LDB0A3-T80.201
DRAM:  16 MB
FLASH:  8 MB
In:     serial
Out:    serial
Err:    serial
Net:    SCC ETHERNET, FEC ETHERNET
PCMCIA:  No Card found
Hit any key to stop autoboot:  5
```

正如你所见，U-Boot 会先打印出版本信息，然后提供与硬件有关的若干细节。U-Boot 引导便会有一个 5 秒的定时器在输出的最后一行开始倒计时。如果在 5 秒内未按下任

何键，U-Boot 就会根据预设的配置引导。如果在 5 秒内按了一个按键，会看到一个提示符：

```
=>
```

你首先可能会想要从 U-Boot 获得辅助说明：

```
=> help
askenv - get environment variables from stdin
autoscr - run script from memory
base - print or set address offset
bdinfo - print Board Info structure
bootm - boot application image from memory
bootp - boot image via network using BootP/TFTP protocol
bootd - boot default, i.e., run 'bootcmd'
cmp - memory compare
coninfo - print console devices and informations
cp - memory copy
crc32 - checksum calculation
date - get/set/reset date & time
dhcp - invoke DHCP client to obtain IP/boot params
diskboot- boot from IDE device
echo - echo args to console
erase - erase FLASH memory
flinfo - print FLASH memory information
go - start application at address 'addr'
help - print online help
ide - IDE sub-system
iminfo - print header information for application image
loadb - load binary file over serial line (kermit mode)
loads - load S-Record file over serial line
loop - infinite loop on address range
md - memory display
mm - memory modify (auto-incrementing)
mtest - simple RAM test
mw - memory write (fill)
nm - memory modify (constant address)
printenv- print environment variables
protect - enable or disable FLASH write protection
rarpboot- boot image via network using RARP/TFTP protocol
reset - Perform RESET of the CPU
run - run commands in an environment variable
saveenv - save environment variables to persistent storage
setenv - set environment variables
sleep - delay execution for some time
tftpboot- boot image via network using TFTP protocol
          and env variables ipaddr and serverip
version - print monitor version
? - alias for 'help'
```

正如所见，U-Boot 有许多命令。幸好，U-Boot 还为每个命令提供了辅助说明：

```
=> help cp
cp [.b, .w, .l] source target count
    - copy memory
```

当 U-Boot 在命令之后附加 `[.b, .w, .l]` 表达式时，表示需要根据要调用的命令版本在命令之后附加相应的字符串。例如，`cp` 命令的三个版本 `cp.b`、`cp.w` 和 `cp.l` 分别可用来复制 `byte`、`word` 和 `long` 类型的数据。

U-Boot 对参数的格式有严格的要求。它会将大部分的参数视为十六进制的数值。以 `cp` 命令为例，代表源地址、目的地址和字节数等参数都必须是十六进制的数值。你不必为这些数值前置或附加任何特殊的符号，例如 `0x` 或 `h`。举例来说，如果源地址是 `0x40000000`，只需键入 `40000000`。

U-Boot 允许你使用代表惟一命令的子字符串来启动该命令。举例来说，如果想要使用 `erase` 命令，可以只键入三个字母 `era`，因为以这三个字母起头的命令只有 `erase`。相对而言，不可以键入 `lo`，因为有三个命令以这两个字母起头：`loadb`、`loads` 和 `loop`。

使用 U-Boot 的环境变量

一旦 U-Boot 启动和执行之后，可以通过设定适当的环境变量来设定它的配置。U-Boot 环境变量的使用与 Unix shell（例如 `bash`）中环境变量的使用非常类似。要检查目标板上环境变量的当前值可以使用 `printenv` 命令。下面是我的控制模块上环境变量的部分内容：

```
=> printenv
bootdelay=5
baudrate=115200
loads_echo=1
serial#=...
ethaddr=00:D0:93:00:05:E3
netmask=255.255.255.0
ipaddr=192.168.172.10
serverip=192.168.172.50
clocks_in_mhz=1
stdin=serial
stdout=serial
stderr=serial

Environment size: 791/16380 bytes
```

每个环境变量的意义各有不同。有些环境变量（例如 `bootdelay`、`serial#` 或 `ipaddr`）已经有缺省的用途。`README` 文件对 U-Boot 的环境变量以及它们的意义有完整的探讨。

如同 Unix shell，可以自行向 U-Boot 加入环境变量。要完成此事，必须使用 `setenv` 命令。

以我的控制模块为例，下面是我自行加入的若干环境变量（即使因为版面的关系它被分成两行，第三条命令也必须以一整行的方式键入）：

```
=> setenv rootpath /home/karim/ctrl-rootfs
=> setenv kernel_addr 40100000
=> setenv nfscmd setenv bootargs root=/dev/nfs rw nfsroot=$(serverip):
    \$(rootpath) ip=$(ipaddr):$(serverip):$(gatewayip):$(netmask):
    \$(hostname)::off panic=1\;
    bootm \$(kernel_addr)
=> setenv bootcmd run nfscmd
```

此例中，我将U-Boot设成从位于0x40100000的内核引导，并且使用NFS来安装它的根文件系统。请注意，我使用\符号告诉U-Boot不应该将\之后的符号视为特殊符号。举例来说，下面是U-Boot所看到的nfscmd：

```
=> printenv nfscmd
nfs2cmd=setenv bootargs root=/dev/nfs rw nfsroot=$(serverip):$(rootpath)
ip=$(ipaddr):$(serverip):$(gatewayip):$(netmask):$(hostname)::
off panic=1;bootm $(kernel_addr)
```

setenv 命令只会为当前会话加入环境变量。因此，如果复位系统，使用*setenv* 设定的任何环境变量都会丢失。为了让环境变量在重引导之后仍能保留，必须将它们存入flash。要完成此事，可以使用*saveenv* 命令：

```
=> saveenv
Saving Enviroment to Flash...
Un-Protected 1 sectors
Erasing Flash...
. done
Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
```

请谨慎使用*saveenv*，因为它将会保存当前定义的所有环境变量，即使是那些临时使用的环境变量。使用*saveenv*之前，请使用*printenv* 检查当前定义的环境变量，确定所保存的都是必要的变量。如果要删除某个变量，只需在不提供任何值的情况下，用*setenv* 定义该变量即可。例如：

```
=> setenv RAMDisk_addr 40500000
=> printenv RAMDisk_addr
RAMDisk_addr=40500000
=> setenv RAMDisk_addr
=> printenv RAMDisk_addr
## Error: "RAMDisk_addr" not defined
```

请注意，= 符号并不会被*setenv* 视为特殊符号。事实上，它会被视为组成环境变量字符串中的一个符号，正如我们在本节前面所见。举例来说，下面的命令是有问题的（与前一个例子中相同的*printenv* 命令相比，此处的*printenv* 命令额外显示了一个=）：


```
=> setenv RAMDisk_addr = 40500000
=> printenv RAMDisk_addr
RAMDisk_addr= 40500000
```

建立引导命令脚本

U-Boot的环境变量可以用来建立引导命令脚本。这种引导命令脚本其实是包含U-Boot命令序列的环境变量。你可以使用`run`命令和`;`（分号）符号的组合让U-Boot执行引导命令脚本。例如我在前一节所设定的环境变量`nfscmd`，实际上是引导命令脚本的一部分。

我在前一节提供的命令脚本之所以能够执行，关键在于`bootcmd`环境变量。系统启动时，U-Boot会将此变量视为自动执行的命令脚本。我将此变量设成`run nfscmd`。换言之，U-Boot应该会在系统启动时执行`nfscmd`命令脚本。环境变量`nfscmd`本身是一组命令。首先它会设定环境变量`bootargs`，U-Boot会将此变量传递给内核作为它的引导参数，然后使用`bootm`命令启动位于`$(kernel_addr)`的内核。分号用来隔开每条命令。`$(VAR_NAME)`符号用来告诉U-Boot将整个字符串替换成环境变量`VAR_NAME`的值。因此当`nfscmd`执行时，`$(kernel_addr)`会被替换成`40100000`（变量`kernel_addr`的值）。同理`$(rootpath)`会被替换成`/home/karim/ctrl-rootfs`。`nfscmd`中其余的环境变量也都会被替换成相应的值。

尽管你可以将`bootcmd`设成包含整个引导命令脚本而不只是使用`run nfscmd`，不过这么做将会很麻烦，倒不如在引导命令行上指定多个命令脚本。你可以通过在`bootcmd`命令脚本中使用`run`命令让U-Boot的环境变量中可以共存多个引导命令行。然后你可以改变系统缺省的引导命令行：

```
=> setenv bootcmd run OTHER_BOOT_SCRIPT
```

或者你可以从命令行直接执行引导命令脚本，而不要改变环境变量`bootcmd`的值：

```
=> run OTHER_BOOT_SCRIPT
```

命令脚本是U-Boot非常有用的特性，不管你是否需要在U-Boot中让某个工作自动化，都应该使用它。

准备二进制映像

因为原始的flash设备并不具备如文件系统一样的结构，也不包含任何形式的文件头，所以把二进制映像加载到目标板时必须携带文件头，好让U-Boot能够区分它们的内容以及知道如何加载它们。前面我们安装的附带在U-Boot套件中的`mkimage`工具程序便是

做这件事的。它可以将 U-Boot 需要的信息加入二进制映像，它还基于验证的目的为二进制映像附加了校验和。

注意：尽管就技术上来说 bootloader 并不需要使用映像头，不过这种映像头在开发期间以及实地应用上有其方便性。因此就被 U-Boot 拿来使用了。

执行 *mkimage* 命令时若不提供任何参数就可以看到它的使用说明：

```
$ mkimage
Usage: mkimage [-l] image
        -l = => list image header information
mkimage -A arch -O os -T type -C comp -a addr -e ep -n name
        -d data_file[:data_file...] image
        -A = => set architecture to 'arch'
        -O = => set operating system to 'os'
        -T = => set image type to 'type'
        -C = => set compression type 'comp'
        -a = => set load address to 'addr' (hex)
        -e = => set entry point to 'ep' (hex)
        -n = => set image name to 'name'
        -d = => use image data from 'datafile'
        -x = => set XIP (execute in place)
```

以我的控制模块为例，下面是我为已编译的 2.4.18 版内核建立 U-Boot 映像的方法：

```
$ cd ${PRJROOT}/images
$ mkimage -n '2.4.18 Control Module' \
> -A ppc -O linux -T kernel -C gzip -a 00000000 -e 00000000 \
> -d vmlinux-2.4.18.gz vmlinux-2.4.18.img
Image Name:   2.4.18 Control Module
Created:      Wed Feb  5 14:19:08 2003
Image Type:   PowerPC Linux Kernel Image (gzip compressed)
Data Size:    530790 Bytes = 518.35 kB = 0.51 MB
Load Address: 0x00000000
Entry Point:  0x00000000
```

这条命令用了相当多的选项，不过你只要参考 *mkimage* 提供的使用说明，就容易了解它们的意义。请注意，*-n* 选项中提供的映像名称不能超过 32 个字符。超过的部分都会被 *mkimage* 忽略掉。这条命令的其余部分告诉 *mkimage*：这是个经过 gzip 压缩的 PPC Linux 的内核映像，加载地址为 0x00000000，而且应该从同一个地址开始执行，输入的映像为 *vmlinux-2.4.18.gz*，输出的 U-Boot 格式映像为 *vmlinux-2.4.18.img*。

RAM disk 的处理采类似的方式：

```
$ mkimage -n 'RAM disk' \
> -A ppc -O linux -T ramdisk -C gzip \
> -d initrd.bin initrd.boot
```

```
Image Name:      RAM disk
Created:         Wed Feb  5 14:20:35 2003
Image Type:      PowerPC Linux RAMDisk Image (gzip compressed)
Data Size:       470488 Bytes = 459.46 kB = 0.45 MB
Load Address:    0x00000000
Entry Point:     0x00000000
```

此例中，参数的个数比较少，因为我们不必指定加载和开始执行的地址。请注意，映像类型已变为 ramdisk。

我们还可以建立一个组合了内核映像和 RAM disk 之 multi 类型的映像。此例中，输入映像包含了两个文件，可以用冒号 (:) 将它们隔开：

```
$ mkimage -n '2.4.18 Ctrl and Initrd' \
> -A ppc -O linux -T multi -C gzip -a 00000000 -e 00000000 \
> -d vmlinux-2.4.18.gz:initrd.bin \
> vmlinux-2.4.18-initrd.img
Image Name:      2.4.18 Ctrl and Initrd
Created:         Wed Feb  5 14:23:29 2003
Image Type:      PowerPC Linux Multi-File Image (gzip compressed)
Data Size:       1001292 Bytes = 977.82 kB = 0.95 MB
Load Address:    0x00000000
Entry Point:     0x00000000
Contents:
  Image 0:       530790 Bytes = 518 kB = 0 MB
  Image 1:       470488 Bytes = 459 kB = 0 MB
```

一旦用 *mkimage* 准备好映像之后，U-Boot 便可将该映像加载到目标板上。正如下面所见，U-Boot 可以使用不同的方法来读取二进制映像。一个方法是使用 Motorola 的 S-Record 格式。如果想使用这个格式，需要进一步处理 *mkimage* 产生的映像，将它转换成 S-Record 格式。我们可以对前面产生的 multi 类型的映像进行 S-Record 格式转换：

```
$ powerpc-linux-objcopy -I binary -O srec \
> vmlinux-2.4.18-initrd.img vmlinux-2.4.18-initrd.srec
```

使用 BOOTP/DHCP、TFTP 和 NFS 引导

如果设好一部服务器为目标板提供 DHCP、TFTP 和 NFS 等服务，正如前文所述，可以从远程启动目标板。以我的控制模块上的 U-Boot 为例，下面示范如何从远程启动我的目标板：

```
=> bootp
BOOTP broadcast 1
DHCP client bound to address 192.168.172.10
ARP broadcast 1
TFTP from server 192.168.172.50; our IP address is 192.168.172.10
Filename '/home/karim/vmlinux-2.4.18.img'.
Load address: 0x100000
```

```

Loading: #####
done
Bytes transferred = 530854 (819a6 hex)

```

bootp 命令会送出服务请求并等待 DHCP 服务器回应。U-Boot 会使用 DHCP 服务器的响应联系 TFTP 服务器，取回 *vmlinux-2.4.18.img* 映像文件，并将它加载到 RAM 中从 0x00100000 开始的地址空间。你可以使用 *iminfo* 命令检查映像的头信息：

```

=> imi 00100000

## Checking Image at 00100000 ...
Image Name: 2.4.18 Control Module
Created: 2003-02-05 19:19:08 UTC
Image Type: PowerPC Linux Kernel Image (gzip compressed)
Data Size: 530790 Bytes = 518.3 kB
Load Address: 00000000
Entry Point: 00000000
Verifying Checksum ... OK

```

正如所见，目标板上 *iminfo* 显示的信息与主机上 *mkinfo* 显示的信息非常类似。针对校验和报告的 OK 字符串代表映像已经正确加载，现在我们可以启动它：

```

> bootm 00100000
## Booting image at 00100000 ...
Image Name: 2.4.18 Control Module
Created: 2003-02-05 19:19:08 UTC
Image Type: PowerPC Linux Kernel Image (gzip compressed)
Data Size: 530790 Bytes = 518.3 kB
Load Address: 00000000
Entry Point: 00000000
Verifying Checksum ... OK
Uncompressing Kernel Image ... OK
Linux version 2.4.18 (karim@Teotihuacan) (gcc version 2.95.3 20010315 ...)
On node 0 totalpages: 4096
zone(0): 4096 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: root=/dev/nfs rw nfsroot= ...
Decrementer Frequency: 5000000
Calibrating delay loop... 79.66 BogoMIPS
...
VFS: Cannot open root device "" or 02:00
Please append a correct "root=" boot option
Kernel panic: VFS: Unable to mount root fs on 02:00
<0>Rebooting in 180 seconds..

```

此例中，内核恐慌是由于它无法找到任何根文件系统。要解决此问题，我们必须使用环境变量建立引导命令脚本，给内核传递适当的引导选项。接下来的命令将会建立一个新的引导命令脚本 *bootpnfs* 并修改缺省的引导命令脚本 *bootcmd*（参见“使用 U-Boot 的环境变量”一节），好让系统能够使用 BOOTP/DHCP、TFTP 和 NFS 来引导：

```
=> setenv bootpnfs bootp\; setenv kernel_addr 00100000\; run nfscmd
=> printenv bootpnfs
bootpnfs=bootp; setenv kernel_addr 00100000; run nfscmd
=> setenv bootcmd run bootpnfs
=> printenv bootcmd
bootcmd=run bootpnfs
```

此例中，bootpnfs 命令脚本会自动执行 *bootp* 命令，以便从 TFTP 服务器取回内核。然后使用 *nfscmd* 命令脚本（建立于“使用 U-Boot 的环境变量”一节）启动此内核。因为此处修改了 *kernel_addr* 变量的值，所以 *nfscmd* 命令脚本会使用 TFTP 加载的内核而不会使用位于 40100000 的内核。

如果现在使用 *boot* 命令，U-Boot 将会完全通过网络来引导。它会通过 TFTP 加载内核，并且通过 NFS 安装根文件系统。如果想保存我们刚才设定的环境变量，请在重新引导之前使用 *saveenv* 命令，否则你必须在引导之后重新设定相同的变量。

将二进制映像加载到 Flash 设备

从网络引导适合在开发初期和测试阶段应用。至于产品的使用，目标板必须将它的内核存入 flash。正如稍后所见，有若干种方法可以用来将内核从主机复制到目标板，并将它存入 flash。然而复制内核映像之前，首先必须设置好用来存储它的 flash 区间，并且抹除这段 flash 区间。以我的控制模块为例，我把缺省的内核存入范围从 0x40100000 到 0x401FFFFFF 的 flash 区间。因此我通过 U-Boot 命令行某除了这段区间：

```
=> erase 40100000 401FFFFFF
Erase Flash from 0x40100000 to 0x401ffffff
..... done
Erased 8 sectors
```

将内核加载到目标板 flash 中的最简单方法，就是先将它加载到 RAM 中，再将它复制到 flash 中。你可以使用 *tftpboot* 命令把内核从主机加载 RAM：

```
=> tftpboot 00100000 /home/karim/vmlinux-2.4.18.img
ARP broadcast 1
TFTP from server 192.168.172.50; our IP address is 192.168.172.10
Filename '/home/karim/vmlinux-2.4.18.img'.
Load address: 0x100000
Loading: ##### ...
done
Bytes transferred = 530854 (819a6 hex)
```

当 *tftpboot* 执行时，它会将 *filesize* 环境变量加入当前的环境变量，并且将它的值设成所下载文件的大小：

```
=> printenv filesize
filesize=819a6
```

你可以在随后的命令中使用这个环境变量来避免手动输入文件的大小。保存环境变量之前别忘了抹除此环境变量，否则它也会被保存起来。

除了 *tftpboot*，还可以使用 *loadb* 命令将映像加载目标板：

```
=> loadb 00100000
## Ready for binary (kermit) download ...
```

此刻，U-Boot 会暂停下来，必须使用主机上的终端仿真程序给目标板送出映像文件。此例中，U-Boot 会根据 *kermit binary* 协议下载数据，因此你必须使用 *kermit* 将二进制映像加载到 U-Boot 中。传送完毕之后，U-Boot 将会输出：

```
## Total Size      = 0x000819a6 = 530854 Bytes
## Start Addr      = 0x00100000
```

此处，U-Boot 也会将 *filesize* 环境变量的值设成所加载文件的大小。正如前面提到的，可能会使用 *iminfo* 命令来检查刚才所加载的映像。

一旦映像存入 RAM 之后，可以将它复制到 flash：

```
=> cp.b 00100000 40100000 $(filesize)
Copy to Flash... done
=> imi 40100000

## Checking Image at 40100000 ...
Image Name:      2.4.18 Control Module
Created:         2003-02-05 19:19:08 UTC
Image Type:      PowerPC Linux Kernel Image (gzip compressed)
Data Size:       530790 Bytes = 518.3 kB
Load Address:    00000000
Entry Point:     00000000
Verifying Checksum ... OK
```

除了“先使用 *tftpboot* 或 *loadb* 将映像加载 RAM 再将它写入 flash”的方法，还可以使用 *loads* 将映像直接加载到 flash 中。在这种情况下，主机会送出 S-Record 格式的映像给目标板。然而与先前两种方法相比，S-Record 文件的下载速度相当慢。在大多数情况下，若改用 *tftpboot* 或 *loadb* 会比较合适（注 6）。

下载 S-Record 格式的文件时，需要使用 *cu* 终端仿真程序把它传送至目标板，因为当其他的终端仿真程序要传送此类文件时，将无法正确地与 U-Boot 交互。通过 *cu* 联机时，可以使用如下的命令：

注 6: *loadb* 命令以及 *tftpboot* 命令默认不会直接加载 flash。尽管 U-Boot 可以在编译期间被设定成使用 *tftpboot* 直接加载 flash，但是并不支持使用 *loadb* 直接加载 flash。

```

=> loads 40100000
## Ready for S-Record download ...
->vmlinux-2.4.18.srec
1 2 3 4 5 6 7 8 9 10 11 12 13 14 ...
...
...176 33177 33178 33179 33180 33181
[file transfer complete]
[connected]

## First Load Addr = 0x40100000
## Last Load Addr = 0x401819A5
## Total Size      = 0x000819A6 = 530854 Bytes
## Start Addr      = 0x00000000

```

此处显示的~>字符串实际上是你必须输入的一部分。它其实是一个`cu`命令，用来发起文件的下载工作。

正如之前所述，一旦映像存入内存，即可以进行检查的工作：

```

-> lmi 40100000

## Checking Image at 40100000 ...
Image Name: 2.4.18 Control Module
Created: 2003-02-05 19:19:08 UTC
Image Type: PowerPC Linux Kernel Image (gzip compressed)
Data Size: 530790 Bytes = 518.3 kB
Load Address: 00000000
Entry Point: 00000000
Verifying Checksum ... OK

```

每当你想把新的映像加载到flash时，必须在事前使用`erase`命令，正如本节一开始所示。

使用 RAM Disk 引导

从RAM disk引导的第一步就是从主机下载RAM disk并且将它安装到目标板的flash中。此处用到的命令如同前一节所示。以我的控制模块为例，下面是完成此事的方法：

```

=> tftpboot 00100000 /home/karim/initrd.boot
ARP broadcast 1
TFTP from server 192.168.172.50; our IP address is 192.168.172.10
Filename '/home/karim/initrd.boot'.
Load address: 0x100000
Loading: ##### ...
done
Bytes transferred = 470552 (72e18 hex)
=> lmi 00100000

## Checking Image at 00100000 ...
Image Name: RAM disk
Created: 2003-02-05 19:20:35 UTC
Image Type: PowerPC Linux RAMDisk Image (gzip compressed)

```

```

Data Size:      470488 Bytes = 459.5 kB
Load Address: 00000000
Entry Point:   00000000
Verifying Checksum ... OK
=> printenv filesize
filesize=72e18
=> imi 40200000

## Checking Image at 40200000 ...
Bad Magic Number
=> erase 40200000 402FFFFFF
Erase Flash from 0x40200000 to 0x402fffff
..... done
Erased 8 sectors
=> cp.b 00100000 40200000 ${filesize}
Copy to Flash... done
-> imi 40200000

## Checking Image at 40200000 ...
Image Name:   RAM disk
Created:      2003-02-05 19:20:35 UTC
Image Type:   PowerPC Linux RAMDisk Image (gzip compressed)
Data Size:    470488 Bytes = 459.5 kB
Load Address: 00000000
Entry Point:  00000000
Verifying Checksum ... OK

```

安装好内核之后，我会用刚才安装的 RAM disk 来启动 flash 中的内核：

```

=> bootm 40100000 40200000
## Booting image at 40100000 ...
Image Name:   2.4.18 Control Module
Created:      2003-02-05 19:19:08 UTC
Image Type:   PowerPC Linux Kernel Image (gzip compressed)
Data Size:    530790 Bytes = 518.3 kB
Load Address: 00000000
Entry Point:  00000000
Verifying Checksum ... OK
Uncompressing Kernel Image ... OK
## Loading RAMDisk Image at 40200000 ...
Image Name:   RAM disk
Created:      2003-02-05 19:20:35 UTC
Image Type:   PowerPC Linux RAMDisk Image (gzip compressed)
Data Size:    470488 Bytes = 459.5 kB
Load Address: 00000000
Entry Point:  00000000
Verifying Checksum ... OK
Loading Ramdisk to 00f2c000, end 00f9edd8 ... OK
Linux version 2.4.18 (karim@Teotihuacan) (gcc version 2.95.3 20010 ...)
On node 0 totalpages: 4096
zone(0): 4096 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line:

```



```
Decrementer Frequency: 5000000
Calibrating delay loop... 79.66 BogoMIPS
...
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
...
RAMDISK: Compressed image found at block 0
...
VFS: Mounted root (ext2 filesystem).
...
```

我们在此处也可以使用环境变量将引导过程自动化。除了可以为内核和RAM disk使用分开的映像，我们还可以使用同时包含此二者的单个映像（就像我们在“准备二进制映像”节所建立的映像）。正如前文所述，U-Boot是个极具弹性的bootloader，具有许多可能的配置。尽管此处的说明无法涵盖所有的可能，不过你可以随意对U-Boot进行试验来获得最适合设置。

从 CompactFlash 设备引导

从CF卡使用U-Boot启动内核之前，需要先插入CF卡并进行适当的分区。你可以使用 *pdisk* 或 *fdisk* 命令（依主机而定）来对CF设备进行分区。因为U-Boot无法辨识任何磁盘文件系统，所以你必须建立若干小分区来存放原始的二进制映像，并用一个大分区来摆放根文件系统，正如我在第七章所说。

以我的控制模块为例，我使用了一张32 MB的CF卡，并且使用 *fdisk* 在上面建立了三个分区：两个2 MB的分区分别用来存放一个稳定版的内核和一个实验版的内核，并且用一个28 MB的分区用来存放我的根文件系统。我会使用 *dd* 命令把内核复制到相应的分区：

```
# dd if=vmlinux-2.4.18.img of=/dev/sda1
1036+1 records in
1036+1 records out
# dd if=vmlinux-2.4.18-preempt.img of=/dev/sda2
1040+1 records in
1040+1 records out
```

我还会使用 *mke2fs* 格式化 */dev/sda3*，将它安装到 */mnt/cf*，并且使用前一章提到的技术将根文件系统复制到 */mnt/cf*。

将CF卡插入使用CF-to-PCMCIA适配卡的PCMCIA端口之后，下面是U-Boot启动时的输出：

```
U-Boot 0.2.0 (Jan 27 2003 - 20:20:21)

CPU:   XPC860xxZPnnD3 at 80 MHz: 4 kB I-Cache 4 kB D-Cache FEC present
Board: TQM860LDB0A3-T80.201
```

```

DRAM: 16 MB
FLASH: 8 MB
In: serial
Out: serial
Err: serial
Net: SCC ETHERNET, FEC ETHERNET
PCMCIA: 3.3V card found: SunDisk SDP 5/3 0.6
        Fixed Disk Card
        IDE interface
        [silicon] [unique] [single] [sleep] [standby] [idle] [low
power]
Bus 0: OK
  Device 0: Model: SanDisk SDCFB-32 Firm: vde 1.10 Ser#: 163194D0310
            Type: Removable Hard Disk
            Capacity: 30.6 MB = 0.0 GB (62720 x 512)
Hit any key to stop autoboot: 5

```

U-Boot 会在启动时发现存储设备。U-Boot 还会提供各种 *ide* 命令让你操作 IDE 存储设备。你可以键入 *help* 命令来检查这些命令：

```

=> help ide
ide reset - reset IDE controller
ide info  - show available IDE devices
ide device [dev] - show or set current device
ide part [dev] - print partition table of one or all IDE devices
ide read  addr blk# cnt
ide write addr blk# cnt - read/write 'cnt' blocks starting at block 'blk#'
                        to/from memory address 'addr'

```

我们可以进一步使用 U-Boot 的命令行来获得更多的设备信息：

```

=> ide part

Partition Map for IDE device 0 -- Partition Type: DOS
Partition      Start Sector      Num Sectors      Type
  1              62              4154              83
  2             4216              4154              83
  3             8370             54312              83

```

这条命令会读取并印出 CF 设备的分区表。此例中，U-Boot 所印出的分区信息符合前面的描述。

要从 CF 设备上某个分区加载内核映像，可以使用 *diskboot* 命令。此命令需要两个参数：内核要加载的地址以及分区识别码。后者是该设备的设备编号和分区编号的组合（以冒号隔开）。下面是把 0 号设备之 1 号分区上的内核映像加载地址 0x00400000 的方法：

```

=> diskboot 00400000 0:1

Loading from IDE device 0, partition 1: Name: hda1
Type: U-Boot
Image Name: 2.4.18 Control Module

```

```

Created:      2003-02-05  19:19:08 UTC
Image Type:   PowerPC Linux Kernel Image (gzip compressed)
Data Size:    530790 Bytes = 518.3 kB
Load Address: 00000000
Entry Point:  00000000
=> imi 00400000

## Checking Image at 00400000 ...
Image Name:   2.4.18 Control Module
Created:      2003-02-05  19:19:08 UTC
Image Type:   PowerPC Linux Kernel Image (gzip compressed)
Data Size:    530790 Bytes = 518.3 kB
Load Address: 00000000
Entry Point:  00000000
Verifying Checksum ... OK

```

内核加载后就可以使用 *bootm* 命令来启动该内核。也可以将环境变量 *autostart* 的值设为 *yes* 让 U-Boot 自动执行此工作。若是后者, *diskboot* 加载内核之后便会自动启动内核:

```

=> setenv autostart yes
=> diskboot 00400000 0:1

Loading from IDE device 0, partition 1: Name: hda1
Type: U-Boot
Image Name:   2.4.18 Control Module
Created:      2003-02-05  19:19:08 UTC
Image Type:   PowerPC Linux Kernel Image (gzip compressed)
Data Size:    530790 Bytes = 518.3 kB
Load Address: 00000000
Entry Point:  00000000
Automatic boot of image at addr 0x00400000 ...
## Booting image at 00400000 ...
Image Name:   2.4.18 Control Module
Created:      2003-02-05  19:19:08 UTC
Image Type:   PowerPC Linux Kernel Image (gzip compressed)
Data Size:    530790 Bytes = 518.3 kB
Load Address: 00000000
Entry Point:  00000000
Verifying Checksum ... OK
Uncompressing Kernel Image ... OK
Linux version 2.4.18 (karim@Teotihuacan) (gcc version 2.95.3 ...)
On node 0 totalpages: 4096
...

```

正如前文所述, 可以通过设定适当的 U-Boot 环境变量让 U-Boot 从 CF 设备引导。此外, 如果希望这么做, 也可以使用 *ide write* 命令从 U-Boot 写入磁盘。欲进一步了解如何使用 U-Boot 的 IDE 能力, 可参考 *help* 命令的输出以及相关文件。

更新 U-Boot

U-Boot 类似于任何其他的开放源码计划：当不断有贡献和补丁被整合进程序代码库时，它会随着时间不断发展。因此你可能会觉得需要更新目标板固件的版本。幸好，因为 U-Boot 是从 RAM 执行的，所以它本身可以用来更新自己。基本上，我们必须将新的版本加载到目标板、抹除旧版的固件并将新版本复制过去。

警告： 这项操作显然具有一定的危险性，只要一点失误或电源出问题都将导致目标板无法引导。因此，当你执行以下步骤时必须特别小心。确定你保存了所要替换的 bootloader 的副本，这样你至少可以恢复到已知可以运行的版本。此外，如果升级失败时你无法使用硬件的方法重新编程目标板的 flash，请认真考虑是否应该避免进行固件的替换。举例来说，假使你没有 BDM 调试器或 flash 编程器，下面的步骤只要有一个失败，都会让你陷入进退两难的窘境。处理有许多缺陷的软件是一回事；变成无法使用的硬件又是另一回事了。

采取了必要的预防措施之后，接着使用 TFTP 将 U-Boot 映像加载到 RAM 中：

```
=> tftp 00100000 /home/karim/u-boot.bin-0.2.0
ARP broadcast 1
TFTP from server 192.168.172.50; our IP address is 192.168.172.10
Filename '/home/karim/u-boot.bin-0.2.0'.
Load address: 0x100000
Loading: #####
done
Bytes transferred = 166532 (28a84 hex)
```

如果并未设置 TFTP 服务器，还可以使用终端仿真程序来传送映像：

```
=> loadb 00100000
## Ready for binary (kermit) download ...

## Start Addr      = 0x00100000
```

与我们之前加载目标板的其他映像不同，无法使用 *imi* 命令来检查映像，因为你加载的 U-Boot 映像并未在主机上经过 *mkimage* 的处理。然而你可以在映像复制到 flash 的前后使用 *crc32* 检查复制结果的正确性。

只有把 U-Boot 存放到未受保护的 flash 区间才可以抹除它（此例中，U-Boot 存放在地址范围从 0x40000000 到 0x4003FFFF 的 flash 区间）：

```
=> protect off 40000000 4003FFFF
Un-Protected 5 sectors
```

抹除先前的 bootloader 映像：

```
=> erase 40000000 4003FFFF
```

```
Erase Flash from 0x40000000 to 0x4003ffff
... done
Erased 5 sectors
```

把新的 bootloader 复制到它的最后目的地:

```
=> cp.b 00100000 40000000 ${filesize}
Copy to Flash... done
```

抹除加载期间设定的 `filesize` 环境变量:

```
> setenv filesize
```

保存环境变量:

```
=> saveenv
Saving Enviroment to Flash...
Un-Protected 1 sectors
Erasing Flash...
. done
Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
```

这个阶段, 已经安装并准备好了可用的新版 bootloader 映像。然而直到执行 *reset* 命令之前, 仍然可以使用目前运行的旧版 U-Boot, 并且可以修正更新期间可能遇到的任何问题。等到更新程序的每个步骤都顺利完成之后, 才可以重新启动系统:

```
=> reset

U-Boot 0.2.0 (Jan 27 2003 - 20:20:21)
CPU:   XPC860xxZPnnD3 at 80 MHz: 4 kB I-Cache 4 kB D-Cache FEC present
Board: TQM860LDB0A3-T80.201
DRAM:  16 MB
FLASH:  8 MB
In:     serial
Out:    serial
Err:    serial
Net:    SCC ETHERNET, FEC ETHERNET
PCMCIA:  No Card found
Hit any key to stop autoboot:  5
```

如果可以再次看到 U-Boot 的引导信息, 代表 U-Boot 已经更新成功。否则代表你更换的固件有问题, 而且你需要使用适当的硬件工具来重新编程 flash 设备。

有时内核映像可以使用较旧版的 bootloader 引导, 却无法使用较新版的 bootloader 引导。举例来说, 当 PPCBoot 从 1.0.5 之前的版本升级到 1.0.5 或之后的版本, 内核在 2.4.5-pre5 之前版本可能会无法引导。问题出在 U-Boot 给内核传递时钟速度值的方法。2.4.5-pre5

之前版本的内核会假设所收到速度值以 MHz 为单位，然而之后版本的内核会假设所收到速度值以 Hz 为单位。PPCBoot 1.0.5 传递给内核的时钟速度值以 Hz 为单位。而内核却假设它所收到速度值以 MHz 为单位，因此无法引导。事实上，引导进程会正常启动，但系统会在 U-Boot 完成映像的解压缩准备启动内核时停住。因此你将会看到如下的情况：

```
...
Entry Point: 00000000
Verifying Checksum ... OK
Uncompressing Kernel Image ... OK
```

之后将不再输出任何信息，也不会响应终端的输入。要解决此问题，需要告诉新版 U-Boot 把以 MHz 为单位的时钟速度值传递给较旧版内核。完成此事的方法就是将 `clocks_in_mhz` 环境变量的值设成 1：

```
=> setenv clocks_in_mhz 1
=> saveenv
```

尽管每次升级的时候不一定会发生此类问题，不过有时候内核的一点改变就会导致与它打交道的工具发生重大变化。如果并未参与每个计划的实际开发，其实很难发现此类问题，强烈建议向该计划的网站订阅 U-Boot users 邮件论坛，不断与其他的 U-Boot 用户接触，并且仔细阅读新版本的发行公告。

设置网络服务



要求嵌入式系统设计者设计的产品具备网络功能的需求正逐渐增多。举例来说，嵌入式系统可能会包含 Web 服务器，并能够基于 Web 进行配置。它也可能会基于维护和更新的目的提供远程登录的功能。因为 Linux 内核以及其上执行的网络软件，经常是用来执行需要高可靠和高可用的网络服务的优选软件，所以你会发现 Linux 特别适合网络应用程序。

本章，我们将会探讨嵌入式 Linux 系统中最常见的网络服务的设置和配置。内容包括如何交叉编译每个网络套件，如何修改目标板的根文件系统以便执行每个套件提供的服务。尤其是，我将会说明 internet super-server 的用法、使用 SNMP 进行远程管理、使用 Telnet 进行网络登录，使用 SSH 进行安全通信，通过 HTTP 提供 Web 服务以及通过 DHCP 进行动态配置。

当然，还有许多其他的网络服务可以在 Linux 上运行。尽管无法以一章的篇幅涵盖所有的网络服务，不过我会提示你如何安装及使用其他的网络套件。此外，我不会说明网络硬件的设置、配置和使用。如果需要了解这方面的信息可以参考 O'Reilly 出版的《Running Linux》和《Linux Network Administrator's Guide》。我也不会仔细说明各个网络套件的配置和使用，因为已经有许多书籍在专门探讨它们。不过以上提到的书籍是以服务器或工作站的观点来探讨 Linux 网络的相关问题。

本章是以第六章的内容为基础。本章所进行的操作其实是第六章提到的建立目标板根文件系统操作的一部分，之所以会分章讨论这些操作，是因为它们并非建立目标板根文件系统的必要操作。

Internet Super-Server

如同大多数其他的 Unix 系统，网络服务在 Linux 中会被实现成监控程序。每个网络监控

程序会响应来自特定端口号的服务请求。例如 Telnet 服务运行在端口号 23 上。为了让网络服务正常运行，必须有个运行中的进程在相应的端口号上监听服务请求。然而大多数系统并不会同时启动所有的网络监控程序来监听相应的端口号，而是使用 *internet super-server*。*super-server* 是个特殊的监控程序，由它负责监听所有已启用的网络服务的端口号。当某个端口号送来服务请求，*super-server* 首先会启动相应的网络监控程序，然后将服务请求传递给该网络监控程序以便提供服务。

这个方案主要有两个好处。首先，因为任何时间只需运行最少的监控程序，所以不会浪费系统资源。其次，可以对网络服务提供集中管理和监控的机制。

尽管 *internet super-server* 可以管理许多网络服务，不过某些服务（例如 HTTP 服务器或 SNMP 代理）基于扩充和可靠的理由几乎总是会被设置成直接控制它们的端口号。事实上，提供此类服务的监控程序不需要通过 *internet super-server* 就能正常运行。接下来将会分节讨论各种网络服务，并且说明该项服务是否与 *super-server* 有依存关系。

Linux 主要有两个 *internet super-server* 可用：*inetd* 和 *xinetd*。尽管 *inetd* 是大多数 Linux 发行套件的标准 *super-server*，不过它有逐渐被功能较多的 *xinetd* 取代的趋势。但因为 *inetd* 的功能不如 *xinetd* 的多，所以 *inetd* 也较小并且较适合嵌入式 Linux 系统使用。

inetd

inetd 是 *netkit* 套件（可以从 <ftp://ftp.uk.linux.org/pub/linux/Networking/netkit/> 获得）的一部分。*netkit* 由各种具备网络能力的套件组成。其中的 *netkit-base* 套件除了附带 *inetd*，还包含 *ping*。如同其他的 *netkit* 套件，*netkit-base* 的发行也是采用 BSD 许可条款。从本节开始直到本章结束，我将会使用一个以 ARM 为基础的系统作为我的系统管理（*system management*，SYSM，注 1）来示范你需要完成的操作。

首先下载 *netkit-base* 并将源码取出放到 `${PRJROOT}/sysapps` 目录。以我的 SYSM 为例，我使用的是 0.17 版的 *netkit-base*。接着移往 *netkit-base* 的源码目录：

```
$ cd ${PRJROOT}/sysapps/netkit-base-0.17
```

开始配置 *netkit-base* 之前需要先修改 *configure* 脚本以免它在主机上执行测试程序。因为你会要求它使用你为目标板建立的编译器，所以它只会将测试程序编译成适合在目标板上执行。因此这些测试程序将无法在主机上执行，而且如果不修改 *configure* 脚本将无法顺利完成工作。要避免这些问题，请编辑 *configure* 脚本将所有企图执行测试程序的每

注 1：欲了解我的范例系统中各组件的相关细节，可参考第一章“以多组件系统为例”一节。

一行注释掉（也就是在该列开头处附加上#符号）。*configure* 实际执行的每个测试程序都称为 *__conftest*。请将如下的每一行：

```
./__conftest || exit 1;
```

修改成：

```
# ./__conftest || exit 1;
```

inetd 的建立需要用到 *glibc* 或 *uClibc*。然而，当你要链接 *uClibc* 时，请确定你有在 *uClibc* 中启用 RPC 的支持。如果 *uClibc* 的建立并未启用 RPC（这是缺省状态），那么你必须重新安装 *uClibc*。

编辑好 *configure* 命令行之后，接着设定和编译 *netkit-base*：

```
$ CC=arm-linux-gcc ./configure --prefix=${TARGET_PREFIX}
$ make
```

netkit-base 的建立相当快。当它以动态方式链接 *glibc* 并经过 *strip* 处理会产生 24 KB 的二进制文件。当它以静态方式链接 *glibc* 并经过 *strip* 处理会产生大约 460 KB 的二进制文件。使用 *uClibc* 时，采用动态链接并经过 *strip* 处理会产生 24 KB 的二进制文件，采用静态链接并经过 *strip* 处理会产生 85 KB 的二进制文件。然而不管你采用何种链接方式，产生的 *inetd* 二进制文件都会比 *xinetd* 二进制文件小很多，我们将在下一节看到。

注意：本章所提供的文件大小是相对于我自己的设置而言，可能会与实际文件大小有些差异。此处提供的数值只具有指示作用，因为二进制文件的大小可能会跟我的不同。例如，一般来说，ARM 二进制码和 RISC 二进制码的大小通常会比 x86 二进制码还大。

与我们在其他章节所建立的其他套件相比，此处并不会使用 *make install*，因为它的 *Makefile* 并不适合应用在跨平台开发。除了别的之外，它的 *Makefile* 还会使用主机的 *strip* 命令删除二进制文件中的符号表。

要安装 *inetd*，请手动把 *inetd* 二进制文件和范例配置文件复制到目标板的根文件系统：

```
$ cp inetd/inetd ${PRJROOT}/rootfs/usr/sbin
$ cp etc.sample/inetd.conf ${PRJROOT}/rootfs/etc
```

请根据你自己的设置来编辑 *inetd.conf* 配置文件。除了 *inetd.conf* 文件，*etc.sample* 目录还包含其他范例文件或许可以使用在目标板的 */etc* 目录，例如 *resolv.conf* 和 *services*。以我的 SYSM 为例，下面是后面将探讨的 *Telnet* 监控程序在 *inetd.conf* 文件中的设定项：

```
telnet  stream  tcp      nowait  root    /usr/sbin/telnetd
```

复制和设定好 *inetd* 之后，接着编辑目标板的 */etc/inittab* 文件，为 *inetd* 加入一行设定。以我的 SYSM 为例，它使用的是 BusyBox 的 *init*，下面是 *inetd* 的设定范例：

```
::respawn:/usr/sbin/inetd -i
```

-i 选项用来要求 *inetd* 不要启动成监控程序。因此，如果 *inetd* 因为某个出乎预料的理由中止了，*init* 可以再次启动 *inetd*（注 2）。

因为 netkit-base 还包含 *ping*，所以你可以在 *ping* 目录中找到 *ping* 二进制文件。然而，如果已经在使用 BusyBox，就不需要使用这个二进制文件，因为 BusyBox 中包含 *ping* 命令。

欲进一步了解 *inetd* 的使用，可参考 netkit-base 套件源码树中 *inetd* 目录包含的 manpage。

xinetd

在某些系统上使用 *xinetd* 会比 *inetd* 还好，因为 *xinetd* 还允许某种安全许可、提供多种登录能力并且能够避免拒绝访问攻击。尽管 *xinetd* 计划网站提供的 FAQ 完整地列出了它优于 *inetd* 的地方，不过只要嵌入式系统被设计用来提供网络服务或是要摆在不安全的网络环境中（例如 Internet）就应该使用 *xinetd* 这个 super-server。

xinetd 由 <http://www.xinetd.org/> 发行，并且采用 BSD-like 的许可条款。以我的 SYSM 为例，我使用的是版本编号 2.3.9 的 *xinetd*。先将 *xinetd* 套件下载并解开放到 *\$(PRJROOT)/sysapps* 目录，然后移往套件源码目录进行其余的操作：

```
$ cd $(PRJROOT)/sysapps/xinetd-2.3.9
```

如同 *inetd*，使用缺乏某些特性的 uClibc 可能无法完成 *xinetd* 的编译。尤其是，如果使用不支持 RPC 和 C99 的 uClibc，*xinetd* 将无法建立完成。除了 C 链接库，*xinetd* 还要依赖 math 链接库（libm）和 cryptography 链接库（libcrypt）。

现在设定、编译以及安装 *xinetd*：

```
$ CC=arm-linux-gcc ./configure --host=$TARGET --prefix=$(TARGET_PREFIX)
$ make
$ make install
```

xinetd 的建立相当快。当它以动态方式链接 uClibc 或 glibc 并经过 strip 处理会产生 130 KB 的二进制文件。当它以静态方式链接 glibc 并经过 strip 处理会产生 615 KB 的二进制

注 2：super-server 通常不容易崩溃。*init* 所提供的只是额外的预防措施。

文件。当它以静态方式链接uClibc并经过strip处理会产生210 KB的二进制文件。*xinetd*套件会把它的组件（包含manpage）安装到`$(TARGET_PREFIX)`目录。*xinetd*二进制文件本身会被安装到`$(TARGET_PREFIX)`。请先将它从该目录复制到目标板的根文件系统，然后使用适当的strip命令来处理它：

```
$ cp $(TARGET_PREFIX)/sbin/xinetd $(PRJROOT)/rootfs/usr/sbin
$ arm-linux-strip $(PRJROOT)/rootfs/usr/sbin/xinetd
```

*xinetd*提供了一个范例配置文件*xinetd/sample.conf*。你可以使用此范例作为你设定目标板配置的基础。请将它复制到目标板的根文件系统，然后根据需要来编辑它：

```
$ cp xinetd/sample.conf $(PRJROOT)/rootfs/etc/xinetd.conf
```

例如下面是我在SYSM的*xinetd.conf*文件中为后面将探讨的Telnet监控程序所设定的配置：

```
service telnet
{
    socket_type          = stream
    wait                 = no
    user                  = root
    server                = /usr/sbin/telnetd
    bind                  = 127.0.0.1
    log_on_failure       += USERID
}
```

最后编辑目标板的*/etc/inittab*文件为*xinetd*加入一行设定。和*inetd*一样，我会在SYSM的*inittab*文件中加入如下的一行设定：

```
::once:/usr/sbin/xinetd
```

与*inetd*不同的是，*xinetd*只能被启动成监控程序。因此，如果*xinetd*中止了，*init*无法再次启动*xinetd*。

欲获得更多关于*xinetd*使用与配置的信息，请参阅*xinetd*套件*xinetd*源码目录中包含的manpage。该计划的网站上还包含FAQ以及邮件论坛。

使用 SNMP 进行远程管理

简易网络管理协议（Simple Network Management Protocol，SNMP）让我们能够通过TCP/IP网络进行设备的远程管理。尽管最可能具备SNMP能力的设备是网络设备（例如路由器和交换器），不过TCP/IP网络上几乎任何设备都会配备SNMP代理（注3）。SNMP

注3：基本上，agent是一个运行在网络设备上的SNMP软件组件，让你能够远程管理网络设备。

代理让你能够在远程对目标板进行自动监控的工作。换言之，操作者不需要站在系统旁边随时注意它是否还运行着以及监控它目前的性能。SNMP代理允许你使用另一个系统上运行的SNMP管理程序（注4）自动询问设备目前的状态。你还可以将目标板上运行的SNMP代理设成在软件或硬件出故障时送出SNMP *trap* 信息通知SNMP manager。如果目标板是一个复杂网络的一部分或者你需要随时从远程监控它的状态，应该考虑将SNMP代理纳入目标板。

有相当多的SNMP代理与套件可以跟具备SNMP能力的设备交互，不过它们的价格多半很贵。在开放源码世界中，Net-SNMP是用来建立和管理具备SNMP能力的系统的标准套件。Net-SNMP以类似于BSD版权的复合型许可条款发行于 <http://net-snmp.sourceforge.net/>（注5）。

Net-SNMP是个相当大的套件，它包含了许多软件组件。然而对大多数目标板来说，我们只对SNMP代理有兴趣，因为这个软件组件让我们能够从远程来管理我们的设备。首先下载Net-SNMP套件并且将它解开放到 `$(PRJROOT)/sysapps` 目录。以我的SYSM为例，我使用的是5.0.6版的Net-SNMP。现在移往套件的源码目录进行其余的操作：

```
$ cd $(PRJROOT)/sysapps/net-snmp-5.0.6
```

你可以使用uClibc或glibc来编译Net-SNMP套件。然而，正如稍后所见，uClibc的使用须符合一些必要条件。除了C链接库，Net-SNMP还依赖“共享对象动态加载”链接库（libdl）以及math链接库（libm）。

想将Net-SNMP的配置设成使用glibc进行建立的工作，请键入：

```
$ CC=arm-linux-gcc ./configure --host=$TARGET --with-endianness=little
```

想让Net-SNMP链接uClibc，需将uClibc设定成支持IPv6。如果不支持，可以在Net-SNMP的配置命令行上加入 `--disable-ipv6` 选项，让Net-SNMP能够停止对IPv6的支持。此外，还需要修改 `agent/mibgroup/ucd-snmp/disk.c` 文件，让使用uClibc的Net-SNMP能够顺利完成编译。请将：

```
#if HAVE_FSTAB_H
    endfsent();
#endif
```

替换成如下的声明：

注4： manager是一个运行在一般工作站或服务上的SNMP软件组件，专门负责监控远程系统上执行的SNMP agent。

注5： 授权条款的细节请参考Net-SNMP套件附带的COPYING文件。

```
#if !defined HAVE_GETMNTENT && defined HAVE_FSTAB_H
    enafsent();
#endif
```

最后使用 *arm-uclibc-gcc*（不是使用 *arm-linux-gcc*）来执行 *configure* 命令。

请注意，设定 Net-SNMP 的配置时应该避免使用 *--prefix* 选项。如果使用了这个选项，所建立的 SNMP 代理会固定前往该选项指定的目录寻找它的文件。事实上，我们会让 SNMP 代理到缺省的 */usr/local/share/snmp* 目录读取它的配置文件。要控制 SNMP 组件的安装目录，我们将会在执行 *make install* 命令时设定 *prefix* 和 *exec_prefix* 的值。

配置命令脚本执行的时候将会提示你键入与 SNMP 代理功能有关的信息，包括你要使用的 SNMP 版本、设备的联络信息以及系统的位置。配置命令脚本提供的指示通常足以让你理解这些必要信息的用途。如果需要获得更多关于 Net-SNMP 代理配置设定过程的信息，请参阅 Douglas Mauro 与 Kevin Schmidt 合著的《Essential SNMP》（O'Reilly）。

配置命令脚本执行完之后，接着建立与安装 Net-SNMP 组件：

```
$ make
$ make prefix=${TARGET_PREFIX} exec_prefix=${TARGET_PREFIX} install
```

我们为变量 *prefix* 和 *exec_prefix* 提供的值，决定了用来安装 Net-SNMP 组件的主目录。避免在配置初期使用 *--prefix* 选项，要在此处设定变量 *prefix* 和 *exec_prefix*，这样尽管 SNMP 代理的组件会被安装到主机上的 *\${TARGET_PREFIX}* 目录，SNMP 代理执行时仍会读取目标板上的 */usr/local/share/snmp* 配置文件。如果忘了设定 *exec_prefix* 将会导致安装失败，因为命令行将会试着把组件安装到主机的 */usr* 目录。

Net-SNMP 建立的 SNMP 代理是一个大型的二进制文件。当它以动态（或静态）方式链接 *glibc* 并经过 *strip* 处理会产生 650 KB（或 1.1 MB）的二进制文件。当它以动态（或静态）方式链接 *uClibc* 并经过 *strip* 处理会产生 625 KB（或 680 KB）的二进制文件。因为任何情况下如果这个代理二进制文件未经 *strip* 处理都会超过 1.7 MB，所以强烈建议以 *strip* 命令来处理这个 agent 二进制文件。

整个建立和安装的过程将会耗费大约 10 分钟的时间，实际所花的时间取决于你使用的硬件，因为 Net-SNMP 是一个相当大的套件。除了复制二进制文件，安装过程中还会将 *manpage* 和头文件复制到 *\${TARGET_PREFIX}* 目录。SNMP 监控程序（*snmpd*）也就是 SNMP 代理会被安装到 *\${TARGET_PREFIX}/sbin*。SNMP trap 监控程序也（用来监控送进来的 trap 信号）会被安装到 *\${TARGET_PREFIX}/sbin*。SNMP 监控程序需要的 MIB 信息会被安装到 *\${TARGET_PREFIX}/share/snmp*。

所有的Net-SNMP组件被安装到主机上的开发工作空间之后,接着把SNMP监控程序安装到目标板的根文件系统:

```
$ cp ${TARGET_PREFIX}/sbin/snmpd ${PRJROOT}/rootfs/usr/sbin
```

然后把`${TARGET_PREFIX}/share/snmp`目录里的相关组件复制到目标板根文件系统中的`/usr/local/share/snmp`目录:

```
$ mkdir -p ${PRJROOT}/rootfs/usr/local/share
$ cp -r ${TARGET_PREFIX}/share/snmp ${PRJROOT}/rootfs/usr/local/share
```

SNMP MIB 信息约有1.3 MB,再加上经过strip处理的二进制文件,整个SNMP套件至少要占用2 MB多一点的存储空间。对大多数嵌入式Linux系统来说,这算是相当大的套件了。

SNMP代理还需要配置文件方能正常执行。Net-SNMP套件建立期间会在套件源码的根目录中产生一个`EXAMPLE.conf`范例配置文件。请依据实际的需要设定此文件,并将此文件复制到`${PRJROOT}/rootfs/usr/local/share/snmp`目录:

```
$ cp EXAMPLE.conf ${PRJROOT}/rootfs/usr/local/share/snmp/snmpd.conf
```

最后编辑目标板的`/etc/inittab`文件,为`snmpd`加入一行设定。以我的SYSM为例,下面是我在它的`inittab`文件中为`snmpd`加入的一行设定:

```
::respawn:/usr/sbin/snmpd -f
```

`-f`选项用来指示`snmpd`不要从调用它的shell衍生出去。换言之,`snmpd`将不会变成监控程序,而且`init`也会在它中止时重新启动它。

若想获得关于SNMP的更多信息(包括Net-SNMP的配置和使用)可参考前面所提到的《Essential SNMP》。Net-SNMP计划的网站包含相当多的资源,包括FAQ、各种文件以及邮件论坛。Net-SNMP所安装的manpage也很有用。

通过 Telnet 进行网络登录

Telnet协议是登录远程网络主机的最简单方法之一。因此,一旦目标板系统连上网络之后,Telnet协议也是存取目标板的最简单方法。要启用远程登录功能,目标板必须执行Telnet监控程序。可以在嵌入式Linux系统上使用的Telnet监控程序主要有两种:`telnetd`(前面所提及的netkit套件的一部分)以及`utelnetd`(由Penguintronix的Robert Schwebel维护)。就文件大小来说,`utelnetd`套件产生的二进制文件显然比netkit Telnet套件产生的二进制文件还小。此外,`utelnetd`不支持internet super-server,`telnetd`则支持internet

super-server。如果系统上的资源非常有限，不要再纳入其他通过 internet super-server 管理的网络服务，请使用 *utelnetd*。

尽管 Telnet 是个既方便又轻量级的通信机制，可让你管理位于专用网络上的设备，不过它不是一个安全的协议，因此不适合在 Internet 上使用。如果需要从远程登录位于 Internet 某处的设备，请改用 SSH。我们将会在第 10 章“使用 SSH 进行安全通信”一节详细讨论 SSH。

netkit-telnetd

如同其他的 netkit 套件，netkit-telnet（内含 *telnetd*）亦采用 BSD 许可条款并且发行于 <ftp://ftp.uk.linux.org/pub/linux/Networking/netkit/>，以我的 SYSM 为例，我使用的是 0.17 版的 netkit-telnet。

下载 netkit-telnet 套件并且将它解压到 `$(PRJROOT)/sysapps` 目录之后，接着移往该套件的源码目录进行其余的操作：

```
$ cd $(PRJROOT)/sysapps/netkit-telnet-0.17
```

正如前面所提到的 netkit-base 套件，netkit-telnet 套件附带的 *configure* 命令脚本会试图执行某些测试程序。因为这些“测试程序”编译时使用的是目标板的编译器，所以将会执行失败。要避免此情况，请编辑 *configure* 命令脚本将试图执行测试二进制文件的每一行注释掉。正如前面所说，请将如下的每一行：

```
./__conftest || exit 1;
```

修改成：

```
# ./__conftest || exit 1;
```

修改好命令脚本之后，可以接着设定和编译 Telnet 监控程序。如果要使用 glibc，请键入：

```
$ CC=arm-linux-gcc ./configure --prefix=$(TARGET_PREFIX)
$ touch $(TARGET_PREFIX)/include/termcap.h
$ make -C telnetd
```

如果要使用 uClibc，请键入：

```
$ CC=arm-uclibc-gcc ./configure --prefix=$(TARGET_PREFIX)
$ touch $(PREFIX)/uclibc/include/termcap.h
$ make -C telnetd
```

正如所见，此处只编译了 *telnetd*。事实上，此套件还包含了 *telnet* 客户程序，然而用户程序的 Makefile 并不支持交叉编译。即使它支持，到头来你将会发现，使用包含在 BusyBox 中的小型 *telnet* 会比较好。此处之所以会使用 *touch* 命令在适当的头文件目录

中产生 *termcap.h* 文件，是因为 *telnetd* 的源码文件中包含这个头文件。然而我们并不需要 *termcap* 链接库。建立过程中只需 *termcap* 头文件存在就行了，所以它可以是一个空的文件。

建立 *telnetd* 的时间相当短。产生的二进制文件相当小。当它以动态（或静态）方式链接 *uClibc* 并经过 *strip* 处理会产生 30 KB（或 65 KB）的二进制文件。当它以动态（或静态）方式链接 *glibc* 并经过 *strip* 处理会产生 30 KB（或 430 KB）的二进制文件。

安装时不要使用 *make install*，因为它的 Makefile 无法正确应用在跨平台的开发上，而且它会试图使用主机的 *strip* 命令（而不是我们前面为目标板所建立的版本）。

改用手动方式把 *telnetd* 二进制文件复制到目标板的根文件系统：

```
$ cp telnetd/telnetd ${PRJROOT}/rootfs/usr/sbin
```

目标板上还需要准备好 *inetd* 或 *xinetd* 经过正确设定的副本以便提供 Telnet 联机的支持。此外，也可以编辑目标板的 */etc/inittab*，使用 *-debug* 选项来启动 Telnet 监控程序，因此它不需要依赖任何 super-server。然而这并非 *telnetd* 缺省的用法。

除了 C 链接库，*telnetd* 还依赖登录例程链接库（*libutil*）。因此，如果要以动态方式链接 *telnetd*，别忘了将 *libutil* 链接库复制到目标板的 */lib* 目录。

关于如何使用 *telnetd* 的进一步信息，请看 *netkit-telnet* 套件的 *telnetd* 目录中包含的 *manpage*，或是主机上或工作站上原生的 *telnetd* 安装的 *manpage*。

utelnetd

采用 GPL 许可条款的 *utelnetd* 套件发行于 http://www.pengutronix.de/software/utelnetd_en.html。*utelnetd* 依赖 C 链接库而且可以使用 *uClibc* 完成建立的工作。以我的 SYSM 为例，我使用的是 *utelnetd* 0.1.3。

下载 *utelnetd* 套件并且将它解压到 *\${PRJROOT}/sysapps* 目录之后，接着移往套件的源码目录进行其余的操作：

```
$ cd ${PRJROOT}/utelnetd-0.1.3
```

utelnetd 不需要在编译之前进行任何配置。要使用 *glibc* 编译此套件，请键入：

```
$ CC=arm-linux-gcc make
```

编译时间非常短，因为整个监控程序就包含在一个源码文件中。当它采用动态链接方式时，不管链接的是 *glibc* 或 *uClibc*，经过 *strip* 处理之后都会产生大约 10 KB 的二进制文件。当它采用静态链接的方式：如果所链接的是 *glibc*，经过 *strip* 处理之后会产生 375

KB 的二进制文件：如果所链接的是 uClibc，经过 strip 处理之后会产生 25 KB 的二进制文件。

utelnetd 不需要任何配置文件。只须将二进制文件复制到目标板的根文件系统，修改系统的初始化过程，让系统在引导期间启动 *utelnetd* 的副本。与 *telnetd* 不同的是，*utelnetd* 是一个独立执行的监控程序，不必依赖 *inetd* 或 *xinetd* 之类的 internet super-server。首先把二进制文件复制到目标板的根文件系统：

```
$ cp utelnetd ${PRJROOT}/rootfs/usr/sbin
```

接着编辑目标板的 */etc/inittab* 文件，让它在系统引导期间启动 *utelnetd*。以我的 SYSM 为例，下面是我在它的 */etc/inittab* 文件中为 *utelnetd* 所加入的一行设定：

```
::respawn:/usr/sbin/utelnetd
```

尽管 *utelnetd* 只附带了一份简单的说明文件 README，不过此套件相当简单，只要将源码文件浏览一遍应该就能获得你所需要的信息。

使用 SSH 进行安全通信

尽管使用 Telnet 就能轻易地与目标板通信，不过它是一个非常不安全的协议，而且各种文件大量报道了它的弱点。例如，用户的密码以明文的方式从客户端传送到服务器。如果在产品中加入 Telnet 监控程序，想要在产品安装到客户的网站之后从远程来修正问题，这是相当不明智的设置而且通常是极其危险的事情。建议最好改用加密能力强的协议，或是可以确保通信机密性的其他机制。目前完成此事的最佳方式就是使用 SSH 协议以及相关的工具程序包。SSH 使用公钥密码学进行端对端通信的加密，并且相当容易使用和部署。

因为 SSH 是个 IETF 标准，目前有若干相互竞争的实现，其中有部分是私有的商业产品。OpenSSH 是主要的开放源码实现。尽管还有其他的开放源码实现，但是它们不是非常难以进行交叉编译，就是它们的依赖性使其不适合在嵌入式 Linux 系统中应用。因此，本节的内容大部分会被用来探讨 OpenSSH。此外，本节将会简述其他的开放源码实现，因为它们终将会演变得可以使用在嵌入式 Linux 系统中。

一个可以通过 SSH 存取的嵌入式系统，通常跟传统服务器一样也会执行 SSH 服务器。因此我们的讨论将会专注在为目标板编译 SSH 服务器、它的设置和配置，以及它在目标板上的使用。不过我将不会说明如何设置以及使用任何其他的 SSH 组件。

如果真的想要在目标板上使用 SSH 套件，建议参考 Daniel Barrett 与 Richard Silverman 合著的《SSH, The Secure Shell: The Definitive Guide》(O'Reilly)。它对此处未提及的部分有深入的探讨。

OpenSSH

OpenSSH 被开发和维护成 OpenBSD 计划的一部分。它采用复合型 BSD 版权（详情参见 *LICENSE* 文件）发行于 <http://www.openssh.org/>，该网站还提供大量的文件以及相当多的其他资源。要在目标板中使用 OpenSSH，还需要两种链接库：OpenSSL 和 zlib。OpenSSL 是 Secure Socket Layer (SSL) 协议的开放源码实现，而且以 BSD-like 版权发行于 <http://www.openssl.org/>。zlib 压缩链接库就是前面我们在第七章“MTD 所支持的设备”一节所探讨的 zlib。建立和安装 OpenSSH 之前，必须先在主机的跨平台开发架构中建立和安装这些链接库。此外，OpenSSH、OpenSSL 和 zlib 也必须以原生的方式存在于主机之上。在继续进行任何其他步骤之前，确定工作站上已经正确安装了必要的组件，因为当你为目标板编译 OpenSSH 的时候需要用到它们。

以我的 SYSM 为例，我使用的是 OpenSSH 3.5p1、OpenSSL 0.9.6g 以及 zlib 1.1。因为这些套件随时都可能被发现攻击弱点，所以务必记下你所使用的套件版本并且准备好升级方案，以便发现这些套件版本存在严重的攻击弱点时能够有所响应。

警告： OpenSSH 套件的交叉编译很麻烦。除了别的之外，它的配置命令脚本会试图执行测试范例，这些测试范例由 `CC=` 指定的编译器建立。因为用来建立应用程序的交叉编译器无法在主机上执行，所以配置命令行会停止执行并送出错误信息。稍后将看到，我们将会利用一些技巧来控制套件的各个配置命令脚本以及用来建立软件的各个 Makefile。举例来说，我们将会使用安装在主机上的原生头文件和链接库来欺骗套件，让它对目标板进行编译的动作。虽然接下来的讨论将会尽可能提供完整的指示，不过你可能需要针对自己的设置花些功夫进行若干修改。

注意： 接下来我会探讨 OpenSSL 和 OpenSSH。因为这两个套件的名称只差一个字母。所以在阅读的时候很容易搞混。阅读的时候请特别注意这两个套件名称的最后一个字母，以免发生混淆。

稍后我们将会讨论如何建立以及安装 OpenSSL。然而在此之前请先参考第七章“MTD 所支持的设备”一节提到的关于如何建立 zlib 的指示。与前面这些指示不同的是，在此处需要将 zlib 建立成静态链接库而不是共享链接库。要完成此事，请不要像前面的指示那样在 *configure* 命令行上设定 `LD_SHARED` 的值，也不要使用 `--shared` 选项。

在这一节其余的说明里，我将会假定你想要使用 glibc 来建立 OpenSSH。如果想要使用 uClibc，请将所有参用到 `$(TARGET_PREFIX)` 的地方改成 `$(PREFIX)/uClibc` 以及将所有参用到 `arm-linux-gcc` 的地方改成 `arm-uclibc-gcc`。安装 uClibc 的时候，如果并未启用影子密码的支持和 C99 的支持，将需要重新安装支持这些特性的 uClibc。此外，还需要

把zlib安装到uClibc的目录。要完成此事，请你在为zlib的安装执行`make install`命令的时候把`prefix`的值设成`${PREFIX}/uClibc`（不要设成`${TARGET_PREFIX}`）。

正确建立和安装好zlib之后，接着下载OpenSSL并将它解开放到`${PRJROOT}/build-tools`目录。然后移往该套件的源码目录，对它进行设定、编译和安装的操作：

```
$ ./config --prefix=${TARGET_PREFIX} compiler:arm-linux-gcc
$ make
$ make install
```

此处，编译器的指定使用`compiler:`选项（不是设定`CC`的值）。以上的命令执行完成之后，所有的组件将会被安装的`${TARGET_PREFIX}`目录。在我设置的硬件上整个编译过程需要耗费大约10分钟的时间。

安装好OpenSSL之后，接着下载OpenSSH并将它解压到`${PRJROOT}/sysapps`目录。然后移往OpenSSH的源码目录继续进行其余的操作：

```
$ cd ${PRJROOT}/sysapps/openssh-3.5p1
```

让OpenSSH的`configure`命令行能够成功产生有用Makefile的技巧就是，假装我们在为目标板产生“建立配置”，但实际上在为主机产生“建立配置”。然后我们使用这些由`configure`产生的Makefile为目标板建立OpenSSH。要想这个方法成功，我们必须假造一些文件链接。我们需要：

1. 为主机的C编译器建立符号链接并以目标板的C编译器命名。
2. 为主机的原生OpenSSL头文件建立符号链接。
3. 为主机的原生OpenSSL链接库建立符号链接。

以我的开发主机为例，原生的OpenSSL头文件和链接库就放在`/usr/local/ssl`目录，C编译器就放在`/usr/bin`。以我的SYSM为例，下面是建立OpenSSH的预备步骤：

```
$ export PATH=./:$PATH
$ which gcc
/usr/bin/gcc
$ ln -s /usr/bin/gcc ./arm-linux-gcc
$ ln -s /usr/local/ssl/include ./fake-include
$ ln -s /usr/local/ssl/lib ./fake-lib
```

我修改`PATh`迫使`configure`先从当前目录开始寻找所有的二进制文件。这让我们得以使用实际上是主机自己的编译器但却被称为`arm-linux-gcc`的编译器。现在我们可以使用这些冒名顶替的链接来执行`configure`命令脚本：

```
$ CC=arm-linux-gcc CFLAGS=-I./fake-include LDFLAGS=-L./fake-lib \
> ./configure --host=${TARGET}
```

上面这条命令会产生能够以动态链接方式建立二进制文件的Makefile。要以静态连接方式建立二进制文件请把LDFLAGS的值变更为“-L./fake-lib -static”。

这个脚本的输出与其他*configure*脚本的输出类似。不过最后它还会打印出所得到的配置摘要：

```
OpenSSH has been configured with the following options:
    User binaries: /usr/local/bin
    System binaries: /usr/local/sbin
    Configuration files: /usr/local/etc
    Askpass program: /usr/local/libexec/ssh-askpass
    Manual pages: /usr/local/man/manX
    PID file: /var/run
    Privilege separation chroot path: /var/empty
    sshd default user PATH: /usr/bin:/bin:/usr/sbin:/sbin:...
    Manpage format: doc
    PAM support: no
    KerberosIV support: no
    KerberosV support: no
    Smartcard support: no
    AFS support: no
    S/KEY support: no
    TCP Wrappers support: no
    MD5 password support: no
    IP address in $DISPLAY hack: no
    Use IPv4 by default hack: no
    Translate v4 in v6 hack: yes
    BSD Auth support: no
    Random number source: OpenSSL internal ONLY
    Host: arm-unknown-linux-gnu
    Compiler: arm-linux-gcc
    Compiler flags: -I./fake-include -Wall -Wpointer-arith -Wno-un...
    Preprocessor flags:
    Linker flags: -L./fake-lib
    Libraries: -lutil -lz -lnsl -lcrypto -lcrypt
```

这段输出内容显示，SSH软件会操作*/usr*和*/var*等根目录，这样并没有什么问题。因为它会以root身份在目标板上执行。然而最重要的部分是位于底部的Compiler字段和各个*flags*字段。如这段输出所示，编译器的名称是对的，而且头文件和链接库的引用路径也都指向前面假造的链接。因此已经成功地瞒过了*configure*，它产生的Makefile会用到我可以控制的文件名称。这个技巧的最后一个步骤就是移除前面假造的链接并且为我的目标板建立适当的链接：

```
$ rm arm-linux-gcc fake-include fake-lib
$ ln -s ${TARGET_PREFIX}/include ./fake-include
$ ln -s ${TARGET_PREFIX}/lib ./fake-lib
```

移除*arm-linux-gcc*文件的链接后，会迫使Makefile使用在PATH路径中找到的*arm-linux-gcc*命令（这就是前面为目标板所建立的交叉编译器）。同样地，我刚才建立的链接库和

头文件的路径链接，会迫使 Makefile 使用目标板实际的链接库和头文件。完成以上设定之后，开始建立 OpenSSH。

如果目标板使用的 C 链接库版本与主机使用的不同，或是目标板上的各种 C 数据类型的大小与主机上的不同，执行 *make* 命令之前，可能需要手动调整某些头文件。以我的例子来说，我必须编辑 *defines.h* 和 *config.h*。在 *defines.h* 文件中，我必须在 `__ss_family` 定义的前后加上 `#if 0` 和 `#endif`。在 *config.h* 文件中，我必须对 `HAVE_GETGROUPLIST` 的 `#define` 做相同的处理。要让 OpenSSH 顺利完成编译，或许需要以自己的方法来修改这些文件。如果链接期间出现丢失符号的错误，或许意味着你必须编辑 *config.h* 将适当的 `HAVE_` 定义注释掉。请注意，如果再次执行 *configure* 命令脚本，先前对 *config.h* 文件所做的任何修改都会丢失。

头文件改好之后，可以开始建立 OpenSSH：

```
$ make
```

在我的硬件上，整个编译过程耗费的时间少于 5 分钟。产生的 SSH 监控程序（正如前面所说，在目标板中这是我们最感兴趣的二进制文件）相当大。当它以动态（或静态）方式链接 `glibc` 并经过 `strip` 处理会产生大约 1 MB（或 1.4 MB）的二进制文件。当它以动态（或静态）方式链接 `uClibc` 并经过 `strip` 处理会产生大约 1 MB（或 1.1 MB）的二进制文件。

现在将 SSH 监控程序复制到目标板的根文件系统，并且对它进行 `strip` 处理：

```
$ cp ./sshd ${PRJROOT}/rootfs/usr/sbin  
$ arm-linux-strip ${PRJROOT}/rootfs/usr/sbin/sshd
```

欲执行此监控程序，需要一个配置文件以及一组私有密钥和公开密钥。OpenSSH 套件附带有范例配置文件 *sshd_config*。请将该文件复制到目标板的根文件系统，并且根据需要定制：

```
$ mkdir -p ${PRJROOT}/rootfs/usr/local/etc  
$ cp sshd_config ${PRJROOT}/rootfs/usr/local/etc
```

此外，还需要为目标板产生密钥。你可以产生三种密钥：RSA1、RSA 和 DSA，每种密钥都包含私有和公开两个部分。所有密钥将会放在跟监控程序的配置文件一样的目录，也就是目标板的 */usr/local/etc/* 目录。现在使用主机的原生 OpenSSH 工具为目标板建立密钥：

```
$ ssh-keygen -t rsa1 -f ${PRJROOT}/rootfs/usr/local/etc/ssh_host_key  
$ ssh-keygen -t rsa -f ${PRJROOT}/rootfs/usr/local/etc/ssh_host_rsa_key  
$ ssh-keygen -t dsa -f ${PRJROOT}/rootfs/usr/local/etc/ssh_host_dsa_key
```

此外，在目标板的根文件系统上为 OpenSSH 建立必要的 */var* 条目：

```
$ mkdir -p ${PRJROOT}/rootfs/var/run ${PRJROOT}/rootfs/var/empty
$ su -m
Password:
# chown root:root ${PRJROOT}/rootfs/var/run ${PRJROOT}/rootfs/var/empty
# chmod 755 ${PRJROOT}/rootfs/var/empty
# exit
```

最后目标板上需要加入一个“不涉及特权的用户”。这个用户可以隔开 SSH 监控程序与外界的连接。因此，如果连接遭到入侵，对方将无法获得系统的 root 存取权限。要加入“不涉及特权的用户”，首先你得在目标板的 */etc/group* 文件中加入如下的一行设定：

```
sshd:x:255:
```

请根据实际情况将 255 替换成适合使用在目标板上的群组标识码。如果使用的是 CRAMFS 文件系统，不要忘了此值必须低于 256。接着你得在目标板的 */etc/passwd* 文件中为“不涉及特权的用户”加入如下的一行设定：

```
sshd:x:501:255:sshd privsep:/var/empty:/bin/false
```

然后你得在目标板的 */etc/shadow* 文件中为“不涉及特权的用户”加入如下的一行设定：

```
sshd:!:11880:0:99999:7:-1:-1:0
```

此外，如果 *sshd* 采用动态链接的方式，还得将 *sshd* 依存的所有链接库全部复制到目标板的根文件系统。此时可以使用 *arm-uclibc-ldd* 找出整个依存关系。

如同前面提到的网络套件，也必须在目标板的 */etc/inittab* 文件中为 *sshd* 启动加入如下的一行设定：

```
::respawn:/usr/sbin/sshd -D
```

-D 选项用来指示 *sshd* 不要从调用它的 shell 衍生出去而成为监控程序。因此，*init* 会在它中止时重新启动它。尽管不是时常发生，但是只要 *sshd* 有任何问题，都应该视为严重的缺陷，立即进行严格的调查。

关于如何设定以及操作 OpenSSH 的进一步信息，请参阅前面所提到的《SSH, The Secure Shell: The Definitive Guide》这本书。

其他的 SSH 实现

除了 OpenSSH，SSH 还存在若干其他的开放源码实现；其中最值得注意的有 LSH 和 FreSSH。然而写作本书时，LSH 和 FreSSH 还不适合在产品级的嵌入式 Linux 系统中使用。

举例来说, LSH 依赖的套件本身还具备自己的依赖关系。尤其是, 它要依赖 GNU MP 链接库、zlib 和 liboop。头两种依赖关系还好。问题出在 liboop 依赖 glib, glib 却需要 pkg-config。此外, glib 套件并不适合进行交叉编译。如果主机使用跟目标板一样架构, 或许你应该考虑以静态链接的方式来编译 LSH, 然后将它使用在目标板上。如果目标板使用跟主机不一样的架构, LSH 就不适用了。

另一方面, FreSSH 是相当新的套件, 因此它的发展不如 SSH 的其他开放源码套件成熟。此外, 它没有 *configure* 命令脚本, 并且需要对 Makefile 进行大规模的修改方能建立完成。此外, 只能使用 glibc 来建立它。要使用 uClibc 进行编译需要对源码进行若干修改。当你使用 glibc 进行编译时, 产生的 SSH 监控程序约有 850 KB, 这跟 OpenSSH 产生的 SSH 监控程序非常接近。

通过 HTTP 提供 Web 内容

具备网络能力的嵌入式系统中最主要的趋势之一就是包含 Web (HTTP) 服务器。这个 HTTP 服务器可用于远程管理或远程数据检查。以我的 SYSM 为例, 这个 HTTP 服务器可以让我的用户设定和监视控制模块的各种状态。

尽管开放源码的 Apache HTTP 服务器是目前世界上用得最多的 HTTP 服务器, 但是它却不适合在嵌入式系统中应用。主要是因为它非常难以进行交叉编译, 而且需要使用大量的存储空间。尽管如此, 仍存在更适合嵌入式系统采用的其他开放源码 HTTP 服务器。尤其是, Boa 和 *thttpd* 等既小又快的轻量级服务器非常适合嵌入式 Linux 系统使用。

似乎无法找出一组明确的特性帮助我们从 Boa 和 *thttpd* 中选出最合适的 HTTP 服务器。惟一值得注意的差别是 Boa 的发行采用的是 GPL 的许可条款, *thttpd* 的发行采用的则是类似 BSD 的许可条款。然而, 它们所产生的二进制文件就大小而言可以一较长短。它们也都支持 CGI 脚本。因此, 建议先对它们有一番了解之后, 再决定哪一个更适合自己。

Boa

Boa 可以从 <http://www.boa.org/> 获得, 它的发行采用的是 GPL 的许可条款。Boa 只需 C 链接库, 而且你可以使用 glibc 和 uClibc 来编译它。以我的 SYSM 为例, 我使用的是 Boa 0.94.13。

首先下载 Boa 并且将它解开放到 `$(PRJROOT)/sysapps` 目录, 接着移往适当的目录:

```
$ cd $(PRJROOT)/sysapps/boa-0.94.13/src
```

然后设定和编译 Boa:

```
$ ac_cv_func_setvbuf_reversed=no CC=arm-linux-gcc ./configure \
> --host=$TARGET
$ make
```

上面的命令脚行将会产生动态链接的二进制文件。使用uClibc进行编译的时候，如果想要产生静态链接的二进制文件，请将LDFLAGS="-static"加入make命令行。使用glibc进行编译的时候，如果想要产生静态链接的二进制文件，请改用如下的make命令行：

```
$ make \
> LDFLAGS="-static -Wl --start-group -lc -lnss_files -lnss_dns \
> -lresolv -Wl --end-group"
```

警告： 如果想使用glibc静态链接Boa，但是你却为链接库设定“建立配置”的时候却没有使用--enable-static-nss选项，那么上面的命令行将会因为丢失文件而执行失败。

如果使用的是建立时未启用静态NSS链接的glibc，为了避免上述错误而只使用LDFLAGS="-static"，正如我在第四章所说，这样所产生的二进制文件将无法正确地运行在目标板上。主要的问题在于，此二进制文件会试图从目标板上的\${TARGET_PREFIX}加载它的动态NSS链接库。因为这个目录并不存在，所以Boa根本就找不到它所需要的链接库。Boa会因而停止执行。尽管它报告的可能是不同的问题，如unknown user，不过你可以用strace来检查它试图打开的文件。

要避免以上这些问题，必须使用--enable-static-nss选项重新编译glibc。一旦重新编译并重新安装链接库之后，将能够以静态链接的方式让二进制文件包含适当的NSS链接库。

请注意，如果使用的是uClibc则不会遇到这种问题，因为uClibc并未实现glibc形式的NSS。

编译时间很短。当它以动态（或静态）方式链接uClibc并经过strip处理会产生60 KB（或90 KB）的二进制文件。当它以动态（或静态）方式链接glibc并经过strip处理会产生60 KB（或520 KB）的二进制文件。

准备好二进制文件之后，接着将它复制到目标板的根文件系统，然后对它进行strip处理：

```
$ cp boa ${PRJROOT}/rootfs/usr/sbin
$ arm-linux-strip ${PRJROOT}/rootfs/usr/sbin/boa
```

执行Boa的时候，目标板的/etc目录中必须有一个boa/子目录，而且boa/子目录必须有个配置文件。现在让我们产生Boa需要的目录，并且将范例配置文件复制过去：

```
$ mkdir -p ${PRJROOT}/rootfs/etc/boa
$ cp ../boa.conf ${PRJROOT}/rootfs/etc/boa
```

Boa的执行需要一个用户账号。这个用户账号在boa.conf文件中指定。请编辑这个文件以及目标板的/etc/passwd和/etc/groups文件，新增一个用户供Boa使用。Boa在目标板的根文件系统上还需要一个用来记录存取和错误状态的/var/log/boa目录：


```
$ mkdir -p ${PRJROOT}/rootfs/var/log/boa
```

注意：切记，如果日志文件的增长没有限制，将会对嵌入式系统造成问题。举例来说，应该要有一个周期性执行的命令行可以重新初始化这类文件；这是确保日志文件不会耗尽可用存储空间的一个简单方法。

Boa执行的时候会到目标板的/var/www目录寻找它的网页内容。你应该将任何的HTML文件（包括*index.html*）放到此处。现在让我们产生此目录并把网页的内容复制过去：

```
$ mkdir -p ${PRJROOT}/rootfs/var/www
$ cp ... ${PRJROOT}/rootfs/var/www
```

最后在目标板的/etc/inittab文件中为Boa加入一行设定。以我的SYSM为例，下面是我为Boa加入的一行设定：

```
::respawn:/usr/sbin/boa
```

关于如何使用Boa的进一步信息，请参考Boa套件附带的文件以及计划网站上提供的文件。

thttpd

thttpd 可以从 <http://www.acme.com/software/thttpd/> 获得，它的发行采用的是 BSD-like 的许可条款。除了 C 链接库，*thttpd* 还依赖 cryptography 链接库（libcrypt）。首先下载 *thttpd* 并将它解压到 *\${PRJROOT}/sysapps* 目录。以我的 SYSM 为例，我使用的是 *thttpd 2.23beta1*。接着移往套件的源码目录以便进行其余的操作：

```
$ cd ${PRJROOT}/sysapps/thttpd-2.23beta1
```

然后设定和编译 *thttpd*：

```
$ CC=arm-linux-gcc ./configure --host=$TARGET
$ make
```

下面这个命令行会产生经动态链接的二进制文件。如同 Boa，要产生经静态链接的二进制文件，必须将 `LDFLAGS="-static"` 加入 *make* 命令行。如同 Boa，要以动态的方式链接 glibc，必须使用如下的 *make* 命令行。

```
$ make \
> LIBS="-static -Wl --start-group -lc -lnss_files -lnss_dns \
> -lcrypt -lresolv -Wl --end-group"
```

警告： 如同Boa一样，如果想以静态方式链接“建立时未启用静态NSS链接的glibc”，如上的make命令行将会执行失败。即使你为了避免这样的错误而使用 `LDFLAGS="-static"`，产生的二进制文件也无法正确地在目标板上运行。详情参见上一节的提示。

如同Boa一样，它的编译时间很短。当它以动态（或静态）方式链接uClibc并经过strip处理会产生70 KB（或115 KB）的二进制文件。当它以动态（或静态）方式链接glibc并经过strip处理会产生70 KB（或550 KB）的二进制文件。

准备好二进制文件之后，接着将它复制到目标板的根文件系统，然后对它进行strip处理：

```
$ cp thttpd ${PRJROOT}/rootfs/usr/sbin
$ arm-linux-strip ${PRJROOT}/rootfs/usr/sbin/thttpd
```

与Boa不同的是，可以使用 `-C` 选项为 *thttpd* 指定配置文件。你可以在 *contrib/redhat-rpm/thttpd.conf* 找到一个范例配置文件。如果希望使用配置文件，请先将范例配置文件复制到目标板的根文件系统，然后依据目标板的配置编辑此文件：

```
$ cp contrib/redhat-rpm/thttpd.conf ${PRJROOT}/rootfs/etc
```

如同Boa一样，*thttpd* 的操作需要一个特殊的用户账号。缺省情况下，它会使用 *nobody* 这个账号。产生此账号的过程已在前面说明过了，要不然你也可以把 *thttpd* 设定成使用你选择的账号。前面复制的范例配置文件，会把 *thttpd* 的用户设成 *httpd* 这个账号，并且将网页的内容放在目标板的 */home/httpd/html* 目录：

```
$ mkdir -p ${PRJROOT}/rootfs/home/httpd/html
```

最后编辑目标板的 */etc/inittab* 文件。我会在SYSM的 *inittab* 文件中为 *thttpd* 加入如下一行设定：

```
::respawn:/usr/sbin/thttpd -C /etc/thttpd.conf
```

关于如何安装以及执行 *thttpd* 的进一步信息，请参考套件附带的文件以及计划网站上提供的文件。

Apache 简介

Apache 可以从 <http://www.apache.org/> 获得，它采用 Apache 的许可条款发行（注6）。正如前文所述，Apache 难以进行交叉编译。如果不顾这样的警告仍想要对 Apache 交叉编译，建议你参考 David McCreedy 于 Apache development 邮件论坛的信件中提到的程序

注6：这是个 BSD-like 的授权条款。完整的授权细节参见套件附带的 *LICENSE* 文件。

(<http://hypermail.linklord.com/new-httpd/2000/May/0175.html>)。如果编译成功了,可能会想要看一下 Ben Laurie 与 Peter Laurie 合著的《Apache: The Definitive Guide》(O'Reilly) 以便进一步了解 Apache 的配置以及使用。

通过 DHCP 进行动态配置

动态主机配置协议 (Dynamic Host Configuration Protocol, DHCP) 可让主机自动进行网络配置。自动配置设定通常会涉及 IP 地址的配置,不过也可能会包括其他的配置参数,正如前面在第九章所见。使用 DHCP 的网络上会存在两种实体:客户端(送出配置请求)以及服务器(为客户端提供可用的配置)。

嵌入式 Linux 系统要成为 DHCP 服务器很容易。在我的范例系统中,SYSM 可以向 UI 提供动态配置服务。反过来说,嵌入式 Linux 系统也可能需要从 DHCP 服务器获得它自己的配置。以我的 UI 模块为例,它便是从 SYSM 获得它的配置。

在大多数 Linux 发行套件中使用的标准 DHCP 套件是来自 Internet Software Consortium (因特网软件联盟,ISC)。尽管这个套件似乎可以在嵌入式 Linux 系统中应用,因为它的应用非常广泛而且同时包含了 DHCP 客户端和 DHCP 服务器,但事实上,它的 Makefile 和配置脚本并不适合进行任何形式的交叉编译。

尽管如此,还存在另外一个同时提供 DHCP 服务器和 DHCP 客户端的开放源码套件 udhcp,它可以被用在嵌入式 Linux 系统中。udhcp 计划是 BusyBox 计划的一部分,它的网站位于 <http://udhcp.busybox.net/>。你可以从该网站获得 udhcp 套件,此套件的发行采用 GPL 的许可条款。udhcp 只依赖 C 链接库,而且你可以使用 glibc 和 uClibc 来编译它。

首先下载 udhcp 套件并将它解压到 `${PRJROOT}/sysapps` 目录。以我的 SYSM 为例,我使用的是 udhcp 0.9.8。接着移往套件的源码目录以便进行其余的操作:

```
$ cd ${PRJROOT}/sysapps/udhcp-0.9.8
```

这个套件并不需要“建立配置”。因此,可以直接编译此套件:

```
$ make CROSS_COMPILE=arm-uclibc-
```

此处的编译时间也很短。如果想要以静态链接的方式建立二进制文件,请将 `LDFLAGS="-static"` 加入 `make` 命令行。如果想使用 glibc 来建立二进制文件,还需要将 `CROSS_COMPILE` 的值设成 `arm-linux-`。当套件以动态方式链接 glibc 并经过 strip 处理(注 7),客户端和服务器的尺寸约有 16 KB。当套件以静态方式链接 glibc 并经过 strip

注 7: udhcp 的 Makefile 一旦完成建立工作会自动对二进制文件进行 strip 处理。

处理，客户端的大小为 375 KB，而服务器的大小为 450 KB。请注意，udhcp 会使用 glibc 的 NSS，因此需要修改 Makefile，以便在编译设定行的结尾（而不要像缺省情况下那样在中间）传递链接选项。如果要以静态方式链接 glibc，还需要像前面建立 Boa 和 *thttpd* 那样来设定 LDFLAGS 的值。当套件以动态（或静态）方式链接 uClibc 并经过 strip 处理，客户端和服务器的尺寸约为 15 KB（或 40 KB）。

如果想在系统中使用服务器，请将它复制到目标板的 */usr/sbin* 目录：

```
$ cp udhcpd ${PRJROOT}/rootfs/usr/sbin
```

如果想使用客户端，请将它复制到目标板的 */sbin* 目录：

```
$ cp udhpcd ${PRJROOT}/rootfs/sbin
```

服务器和客户端都需要配置文件以及运行时文件来保存与租约状态有关的信息。

对服务器来说，需要产生 */var/lib/misc* 目录和租约文件，以及把范例配置文件复制到目标板的根文件系统：

```
$ mkdir -p ${PRJROOT}/rootfs/var/lib/misc
$ touch ${PRJROOT}/rootfs/var/lib/misc/udhcpd.leases
$ cp samples/udhcpd.conf ${PRJROOT}/rootfs/etc
```

如果忘了产生租约文件，服务器将拒绝启动。

对客户端来说，需要产生 */etc/udhpcd* 目录和 */usr/share/udhpcd* 目录，以及把某个范例配置文件复制到 */usr/share/udhpcd/default.script*：

```
$ mkdir -p ${PRJROOT}/rootfs/etc/udhpcd
$ mkdir -p ${PRJROOT}/rootfs/usr/share/udhpcd
$ cp samples/sample.renew \
> ${PRJROOT}/rootfs/usr/share/udhpcd/default.script
```

此外，还得编辑目标板的 */etc/inittab* 文件，让它按照需要启动监控程序。以我的 SYSM 为例，我为 DHCP 服务器加入了如下一行设定：

```
::respawn:/usr/sbin/udhcpd
```

至于 udhcpd 和 udhpcd 配置和使用的细节，请阅读套件附带的 manpage 并访问该计划的网站。



第十一章

调试工具

前一章已经讨论过了如何设置、配置以及使用各种既有的自由和开放源码的软件组件。现在你正准备测试系统，还需要若干有用的调试工具。

这一章，我们将讨论嵌入式Linux系统在开发期间使用的主要软件调试工具的安装和使用。这项讨论涵盖用gdb进行应用程序的调试，跟踪应用程序和系统的行为、性能分析以及内存调试。此外，我还会简述嵌入式Linux系统在开发期间常会用到的硬件工具。因为目标板上不同的操作系统使用硬件调试工具的方式各有一些差异，所以此处并不会讨论如何使用它们。尽管如此，我将会对“使用硬件工具对嵌入式Linux系统中运行的软件进行调试”的各种方法提出建议。

要能善用本章提到的工具，强烈建议让目标板使用通过NFS安装的根文件系统。一旦找到并修正缺陷之后，这除了可让你快速更新软件，还可以加快调试的速度，因为在你手动给目标板传送更新的二进制文件之后，还会对刚更新的软件进行调试。基本上，一个通过NFS安装的根文件系统可以简化更新和调试的程序，因此可以缩短开发时间。此外，NFS可让你在主机上立即取得目标板产生的数据。

本章将会说明自由和开放软件的调试工具，但由于篇幅有限无法涵盖可在Linux中使用的所有调试工具。尽管如此，本章的内容应该可以帮助你善用在网站或发行套件上能够找到的任何额外的Linux调试工具。请注意，此处的讨论并未涵盖用来进行内核调试的任何工具。如果需要对内核进行调试请参考《Linux Device Drivers》第四章。

用gdb进行应用程序调试

GNU debugger (GNU 调试器，gdb) 是GNU计划的符号调试器，一般认为它是Linux系统上最重要的调试工具。gdb自问世至今已经有10多年的历史了，许多非Linux的

嵌入式系统已经在使用 *gdb* 与所谓的 *gdb stub* 对目标板进行远程调试（注1）。然而，因为 Linux 内核实现了 `ptrace()` 系统调用，所以当你从远程对嵌入式应用程序进行调试时并不需要 *gdb stub*。你可以改用 *gdb* 套件提供的 *gdb* 服务器。此服务器是一个运行在目标板上的小型应用程序，它会执行“运行在主机上的 *gdb* 调试器”送来的命令。因此，不必在目标板上执行 *gdb* 调试器，就可以对目标板运行的任何应用程序进行调试。这一点非常重要，因为后面将会看到，实际的 *gdb* 二进制文件相当大。

这一节将会按照主机 / 目标板的配置来探讨 *gdb* 的安装与使用，但是不包含如何实际地使用 *gdb* 来进行应用程序的调试。想要学习如何设断点、检查变量以及检查以往的跟踪记录，可以阅读内容涉及 *gdb* 使用的书籍或使用手册。尤其是，可以看一下《Running Linux》（O'Reilly）的第十四章，以及 *gdb* 套件附带（与 <http://www.gnu.org/manual/> 下在线所提供）的 *gdb* 使用手册。

建立与安装 *gdb* 组件

gdb 套件可以从 <ftp://ftp.gnu.org/gnu/gdb/> 取得，它是采用 GPL 的许可条款发行的。下载 *gdb* 套件之后将源码取出并放到 `${PRJROOT}/debug` 目录。以我的控制模块为例，我使用的是 5.2.1 版的 *gdb*。如同我在第四章提到的其他 GNU 工具集组件，最好不要使用套件的源码目录来建立实际的调试器。因此，我会另外产生一个“建立目录”并且移往该目录进行 *gdb* 的建立工作：

```
$ mkdir ${PRJROOT}/debug/build-gdb
$ cd ${PRJROOT}/debug/build-gdb
$ ../gdb-5.2.1/configure --target=$TARGET --prefix=${PREFIX}
$ make
$ make install
```

这些命令建立的 *gdb* 调试器可用来处理目标板应用程序。如同其他的 GNU 工具集组件，调试器二进制文件的名称取决于目标板。例如我的控制模块的调试器二进制文件叫做 *powerpc-linux-gdb*。这个二进制文件以及其他的调试器文件会被安装到 `$PREFIX` 目录。在我的硬件上整个建立过程大约要耗费 5 到 10 分钟，所产生二进制文件相当大。以我的 PPC 目标板为例，如果采用动态链接，调试器二进制文件经过 `strip` 处理之后仍有 4 MB。因此 *gdb* 二进制文件无法在目标板上使用，必须改用 *gdb* 服务器。

注1：*gdb stub* 是目标板的固件或操作系统内核里的一组 `hook`（钩子）和 `handler`（处理例程），可用来与远程调试器交互。*gdb stub* 的使用可参考 *gdb* 使用手册的说明。

注意：写作本书时，为目标板建立的 *gdb* 还不能处理目标板的内存转储文件。有问题的程序必须在目标板上执行，并且改用 *gdb* 服务器以原生的方式捕捉其发生的错误。*gdb* 邮件论坛上讨论到“为 *gdb* 加入跨平台内存转储文件的读取能力”这方面的主题，而且已经有若干补丁可用。所以当你阅读到此处时，*gdb* 可能已经支持读取跨平台内存转储文件的功能。

前面并未建立 *gdb* 服务器，因为你必须使用适当的工具对它进行目标板的交叉编译。要完成此事，首先得产生一个用来建立 *gdb* 服务器的目录，然后移往该目录进行建立 *gdb* 服务器的工作：

```
$ mkdir ${PRJROOT}/debug/build-gdbserver
$ cd ${PRJROOT}/debug/build-gdbserver
$ chmod +x ../gdb-5.2.1/gdb/gdbserver/configure
$ CC=powerpc-linux-gcc ../gdb-5.2.1/gdb/gdbserver/configure \
> --host=${TARGET} --prefix=${TARGET_PREFIX}
$ make
$ make install
```

现在 *gdb* 服务器二进制文件 *gdbserver* 已经被安装到 *\${TARGET_PREFIX}/bin* 目录。如果采用动态链接，*gdbserver* 经过 *strip* 处理之后还剩下 25 KB。与 *gdb* 相比，*gdbserver* 的大小对目标板适合多了。

建立好 *gdbserver* 之后，接着把它复制到目标板的根文件系统：

```
$ cp ${TARGET_PREFIX}/bin/gdbserver ${PRJROOT}/rootfs/usr/bin
```

在目标板上使用 *gdb* 服务器并不需要任何额外的配置步骤。我将会在下一节说明 *gdb* 服务器的用法。

使用 *gdb* 组件进行目标板应用程序的调试

在可以使用 *gdb* 进行应用程序调试之前，需要使用适当的选项来编译应用程序。你主要应该为 *gcc* 命令行加入 *-g* 选项。此选项会为编译器产生的目标文件加入调试信息。如果想要加入更多调试信息，可以使用 *-ggdb* 选项。然后你可以在应用程序的二进制文件中找到这两个调试选项所加入的信息。尽管这么做会导致二进制文件变得比较大，不过你仍然可以在目标板上使用“经 *strip* 处理”（亦即去掉调试信息）的二进制文件。假定你在主机上备有原本“未经 *strip* 处理”（亦即包含调试信息）的版本。若要这么做，请先在主机上建立包含完整调试信息的应用程序，接着将产生的二进制文件复制到目标板的根文件系统，然后使用 *strip* 命令将这个二进制文件副本中的所有符号信息（包括调试信息）移除。目标板上使用的是“经 *strip* 处理”的二进制文件与 *gdbserver*。主机上使用

调试信息、符号表以及 strip

大多数现代的Linux二进制文件都是采用ELF格式。如同其他格式的二进制文件一样，ELF二进制文件中包含若干区段，这些区段的名称和作用各不相同。二进制文件中实际的执行码通常放在.text区段。还有其他区段，例如：.data用来放经初始化（有初值）的数据，.bss区段用来放未经初始化（无初值）的数据；调试信息通常会放在.stab和.stabstr区段。这些区段将会依据符号表（symbol table，Stabs）调试格式进行编排，其中包含行号、源码文件的路径、头文件的路径、变量声明、型别声明等等信息。

*objdump*和*readelf*都可用来检查ELF二进制文件的各个区段。下面展示了用*readelf*检查未经strip处理的*gdbserver*二进制文件的输出范例：

```
$ powerpc-linux-readelf -S gdbserver
There are 32 section headers, starting at offset 0xlaca4:

Section Headers:
  [Nr] Name                Type              Addr             Off             Size             ES Flg Lk Inf Al
  ...
  [12] .text                 PROGBITS          10000b48          000b48          003008          00  AX  0  0  4
  ...
  [17] .data                 PROGBITS          10015470          005470          000914          00  WA  0  0  4
  ...
  [25] .bss                  NOBITS            10016114          005e54          00236c          00  WA  0  0 16
  [26] .stab                 PROGBITS          00000000          005e54          00798c          0c           27  0  4
  [27] .stabstr              STRTAB            00000000          00d7e0          00d1d5          00           0  0  1
  [28] .comment              PROGBITS          00000000          01a9b5          0001ee          00           0  0  1
  [29] .shstrtab             STRTAB            00000000          01aba3          0000ff          00           0  0  1
  [30] .symtab               SYMTAB            00000000          01b1a4          000f40          10           31 6e  4
  [31] .strtab               STRTAB            00000000          01c0e4          000d19          00           0  0  1
Key to Flags: W (write), A (alloc), X (execute), M (merge), S (strings)
               I (info), L (link order), O (extra OS processing required)
               o (os specific), p (processor specific) x (unknown)
```

*gdbserver*二进制文件经过*strip*处理之后，包含调试信息的区段.stab和.stabstr将会连同.symtab和.strtab一起被移除，其余区段（.shstrtab除外）将会维持不变。其中惟一会变动的区段就是.shstrtab（区段头部字符串表），它的尺寸会变小，因为二进制文件中的区段数目变少了。下面展示了以*readelf*检查经strip处理的*gdbserver*二进制文件的输出范例：

```
$ powerpc-linux-readelf -S gdbserver
There are 28 section headers, starting at offset 0x6134:
```

—待续—

Section Headers:										
[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
...										
[12]	.text	PROGBITS	10000b48	000b48	003008	00	AX	0	0	4
...										
[17]	.data	PROGBITS	10015470	005470	000914	00	WA	0	0	4
...										
[25]	.bss	NOBITS	10016114	005e54	00236c	00	WA	0	0	16
[26]	.comment	PROGBITS	00000000	005e54	0001ee	00		0	0	1
[27]	.shstrtab	STRTAB	00000000	006042	0000f0	00		0	0	1
Key to Flags: W (write), A (alloc), X (execute), M (merge), S (strings)										
I (info), L (link order), O (extra OS processing required)										
o (os specific), p (processor specific) x (unknown)										
欲取得二进制文件格式（包括ELF）的更多信息，可参考John Levine的著作《Linkers & Loaders》（Morgan Kaufmann）。欲取得Stabs格式的信息，可参考gdb套件的gdb/doc目录中（与 http://sources.redhat.com/gdb/current/online/docs/stabs.html 在线）提供的stabs调试格式手册。										

的是“未经strip处理”的二进制文件与gdb。尽管这两个gdb组件使用的是不同的二进制映像，但是执行在主机上的目标板gdb却能够为应用程序找到和使用适当的调试符号，因为它所读取的是“未经strip处理”的二进制文件。

下面是“命令监控程序”的Makefile根据调试的需要所做的变更（原本的Makefile请参考第四章“使用工具链”一节）：

```
...
DEBUG          = -g
CFLAGS         = -O2 -Wall $(DEBUG)
...
```

尽管gcc允许我们同时使用-g和-O这两个选项，不过当你为二进制文件产生调试信息的同时最好不要使用-O选项，因为经过优化的二进制文件与应用程序最初的源码相比可能会有些微妙的差异。举例来说，某些未用到的变量可能不会放入二进制文件，这可能会导致二进制文件中指令实际的执行顺序跟源码中缺省的顺序不同。

目标板上运行的gdb服务器与主机上运行的gdb调试器有两种通信方式：使用“交叉串行连接”或TCP/IP联机。尽管这两种通信接口在许多方面都有所不同，但是你所要执行的命令的语法却非常相似。要发起一项使用gdb服务器的“调试会话”有两个步骤：首先在目标板上启动gdb服务器，接着从主机上的gdb调试器连接至该服务器。

一旦准备进行应用程序的调试，用来在目标板上启动gdb服务器的命令行还包括“通信方式”和“应用程序名称”这两个参数。如果目标板有TCP/IP接口可用，可以在gdb服务器启动时将它设成采用TCP/IP的通信方式：

```
# gdbserver 192.168.172.50:2345 command-daemon
```

此例中，主机的 IP 地址（注 2）是 192.168.172.50，而且主机上用来监听 *gdb* 联机的端口号是 2345。请注意，*gdb* 用来进行主机与目标板间通信的协议并不包含任何形式的认证或安全机制。因此不建议通过公共的 Internet 来进行这种方式的应用程序调试。如果需要以这种方式进行应用程序的调试，可能应该考虑使用 SSH 端口转发功能来加密 *gdb* 会话。《SSH, The Secure Shell: The Definitive Guide》(O'Reilly) 提到如何实现 SSH 端口转发功能。

正如前文所述，转发给 *gdbserver* 的命令监控程序可以是一个“经 strip 处理”的副本，在主机上建立的“未经 strip 处理”的命令监控程序是它的原本。

如果想使用程序连接对目标板进行调试，可以在目标板上使用如下的命令行：

```
# gdbserver /dev/ttyS0 command-daemon
```

此例中，目标板对主机的程序连接位于第一个串行端口 */dev/ttyS0*。

一旦 *gdb* 服务器在目标板上启动之后，在主机上只要使用 *target remote* 命令就可以从 *gdb* 调试器连接至该服务器：

```
$ powerpc-linux-gdb command-daemon
(gdb) target remote 192.168.172.10:2345
Remote debugging using 192.168.172.10:2345
0x10000074 in _start ()
```

此例中，目标板的 IP 地址是 192.168.172.10，而且此处所指定的端口号跟我们之前在目标板上启动 *gdb* 服务器时使用的端口号一样。与目标板上的 *gdb* 服务器不同的是，此处所使用的命令监控程序必须是“未经 strip 处理”的二进制文件。否则进行应用程序调试的时候 *gdb* 就起不了作用。

如果目标板上的程序重新启动，主机上的 *gdb* 并不需要重新启动。一旦目标板上的 *gdbserver* 重新启动之后，只需重新执行 *target remote* 命令即可。

如果主机通过串行连接至目标板，可以使用如下的命令：

```
$ powerpc-linux-gdb progname
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
0x10000074 in _start ()
```

注 2： 写作本书时，*gdbserver* 实际上会忽略此字段。

尽管此例中目标板和主机都是使用 */dev/ttyS0* 彼此互连，不过这只是巧合。目标板和主机可以使用不同的串行端口彼此互连，它们指定的是当地接有串行线的串行端口。

目标板和主机连通之后，可以设置断点，并且做你可以在符号调试器中进行的任何事。

对嵌入式目标板进行调试时，可能会发现有若干 *gdb* 命令特别有用，例如：

file

设定欲调试的二进制文件的文件名。从该文件加载调试符号。

dir

为应用程序的源码文件在搜索路径中加入一个目录。

target

设定参数以便连接远程的目标板，正如稍早所说。事实上这并非只是一个命令而是一组命令。欲知详情请使用 *help target* 命令。

set remotebaud

通过串行线对远程应用程序进行调试时用来设定串行端口的速度。

set solib-absolute-prefix

设定搜索路径以便找到欲调试的二进制文件使用的共享链接库。

如果使用的是动态链接的二进制文件，最后一个命令可能最有用。在目标板上运行的二进制文件会从 */*（根目录）开始寻找它的共享链接库，但是运行在主机上的 *gdb* 并不知道要如何确定这些共享链接库的位置。你在主机上需要使用如下的命令告诉 *gdb* 去何处寻找正确的目标板二进制文件：

```
(gdb) set solib-absolute-prefix ../../tools/powerpc-linux/
```

与一般的 *shell* 不同的是，*gdb* 命令行并不认得 *\$(TARGET_PREFIX)* 之类的环境变量。因此你必须提供完整的路径。此例提供的路径是运行 *gdb* 当时所在目录的相对路径，不过你也可以使用绝对路径。

如果想在 *gdb* 每次启动时自动执行若干命令，可以使用 *.gdbinit* 文件。这类文件的使用说明可参考 *gdb* 使用手册的“Canned Sequences of Commands”这一节的“Command files”这一小节。

若想取得各种调试器命令的使用信息，可以在 *gdb* 作业环境中使用 *help* 命令，或是参考 *gdb* 使用手册。

使用图形接口

许多开发者发现使用这种简单的 *gdb* 命令行进行调试既麻烦也不直观。这类开发者很幸运，目前存在不少的图形接口可以将 *gdb* 大部分的复杂性隐藏起来，并且为断点的设定、变量的检查和其他常见的调试工作提供容易使用的机制。这类图形接口（前台程序）的例子包括 DDD (<http://www.gnu.org/software/ddd/>)、KDevelop 以及第四章提到的其他 IDE。与主机上的调试器非常像的是，我们前而为目标板建立的跨平台 *gdb* 很有可能会被你喜爱的调试接口采用。每种前台用来指定调试器二进制文件名称的方法各不相同。详情参见前台程序的说明文件。以我的控制模块为例，我需要设定前台程序，让它使用 *powerpc-linux-gdb* 调试器。

跟踪

符号调试器对寻找和修正程序错误很有用。然而，如果问题涉及应用程序之间或应用程序与内核之间的交互，那么符号调试器能够提供的帮助就很有有限了。要解决这种与行为有关的问题，必须对应用程序与其他软件组件间的实际交互进行跟踪。

跟踪的最简单形式就是监控单一应用程序与 Linux 内核的交互。这让你能够轻易观察到因为传递的参数或系统调用顺序错误导致的任何问题。

然而，以隔离的方式观察单一的进程并无法满足各种需要。例如，如果想要对进程间的同步问题或时效性问题进行调试，将会需要一个涵盖全系统的跟踪机制，它能够精确地提供系统上各个事件发生的顺序和时间。举例来说，为了了解火星上的“火星拓荒者号”为什么会不断重新引导，火箭推进实验室的工程师们会借助 VxWorks 操作系统的系统跟踪工具（注 3）。

还好，单进程跟踪和系统跟踪在 Linux 中都有支持。接下来让我们分节探讨这两种跟踪方式。

单进程跟踪

strace 是单进程跟踪的主要工具。*strace* 会使用 *ptrace()* 系统调用来拦截应用程序用到的任何系统调用。因此，它可以取出所有系统调用的信息，并使用人类可阅读的格式加以显示，让你分析这些信息。因为 *strace* 是个使用广泛的 Linux 工具，所以我并不会说

注 3： 你可以在 http://research.microsoft.com/~mbj/Mars_Pathfinder/Authoritative_Account.html 看到 Glenn Reeves 对此事极具教育性和娱乐性的报导。Glenn 是“火星拓荒者号”软件的首席开发者。

明它的用法，只会说明如何将它安装到目标板。如果想了解 *strace* 的使用细节，可参考《Running Linux》的第十四章。

strace 可以从 <http://www.liacs.nl/~wichert/strace/> 取得，它的发行采用的是 BSD 的许可条款。以我的控制模块为例，我使用的是 4.4 版的 *strace*。下载套件并将它解压到 `${PRJROOT}/debug` 目录之后，接着移往套件的源码目录，然后设定和建立 *strace*：

```
$ cd ${PRJROOT}/debug/strace-4.4
$ CC=powerpc-linux-gcc ./configure --host=$TARGET
$ make
```

如果希望 *strace* 以静态的方式链接 uClibc，请将 `LDFLAGS="-static"` 加入 *make* 命令行。假定 *strace* 要使用 NSS，如果希望 *strace* 以静态的方式链接 glibc，需要使用一条特别的命令行，正如我们在第十章为其他套件所做的那样。

```
$ make \
> LDFLAGS="-static -Wl --start-group -lc -lnss_files -lnss_dns \
> -lresolv -Wl --end-group"
```

当 *strace* 以动态（或静态）方式链接 glibc 并经过 strip 处理会产生 145 KB（或 605 KB）的二进制文件。当 *strace* 以动态（或静态）方式链接 uClibc 并经过 strip 处理会产生 140 KB（或 170 KB）的二进制文件。

建立好二进制文件之后，接着将它复制到目标板的根文件系统：

```
$ cp strace ${PRJROOT}/rootfs/usr/sbin
```

在目标板上使用 *strace* 并不需要额外的配置步骤。此外，*strace* 在目标板上的用法跟在一般的 Linux 工作站上一样。如果需要关于 *strace* 用法的更多信息，可参考前面所提到的网页或是套件附带的 manpage。

系统跟踪

Linux Trace Toolkit (LTT) 是 Linux 主要的系统跟踪工具程序，它的提出者和后续的维护者就是本书的作者。与其他的跟踪工具程序（例如 *strace*）相比，LTT 并不会使用 `ptrace()` 机制来拦截应用程序的行为。LTT 会提供内核补丁，让你能够使用内核监视器监控关键的内核子系统。内核监视器产生的数据首先由跟踪子系统负责收集，然后由跟踪监控程序将这些数据写入磁盘。整个过程对系统的行为和性能的影响非常小。经过大量测试得知，当这个跟踪措施闲置时对系统的影响微乎其微，而且即使在负荷最重的情况下对系统的影响也不会高于 2.5%。

除了可以使用跟踪期间产生的数据重构系统的行为，LTT提供的用户工具程序还可以让你取得跟踪期间与系统行为有关的性能数据。下面摘要列出 LTT 的若干功能：

- 对进程间的同步问题进行调试
- 了解应用程序、系统中其他应用程序以及内核之间的交互
- 测量内核为应用程序请求提供服务花费的时间
- 测量应用程序因为等待较高优先权的其他进程花费的时间
- 测量中断影响整个系统花费的时间
- 了解系统对外界输入的确切反应

为了提供以上功能，LTT 的操作被分成四种软件组件：

- 内核监视器：用来产生你所要跟踪的事件
- 跟踪子系统：用来将内核监视器产生的数据收集放到一个缓冲区里
- 跟踪监控程序：用来将跟踪子系统的缓冲区写入磁盘
- 可视化工具：系统跟踪后的处理，它会使用人类可阅读的形式来显示收集到的数据

头两个软件组件被实现成内核补丁的形式、最后两个组件则是独立的用户层工具。头三个软件组件必须在目标板上运行，最后一个组件（可视化工具）则可运行在主机上。0.9.5a 和之前版本的 LTT，跟踪子系统的存取通过把它当成用户空间的设备，通过适当的 */dev* 条目进行。然而，自开发版 0.9.6 开始，采用内核开发者的建议，去掉了这个抽象层。因此，尽管接下来的说明会把跟踪子系统视为一个设备，不过较新版的 LTT 将不会使用这个抽象层，所以你不必在目标板的根文件系统上产生任何的 */dev* 条目。

因为 LTT 可以侦测和处理具有不同字节顺序的跟踪数据，所以你可以在完全不同的系统上产生并读取跟踪数据。举例来说，x86 主机可以直接读取基于 PPC 的控制模块产生的跟踪数据。

除了可以跟踪一组预先定义的事件，LTT 还允许你自己定制和登录来自用户空间和内核空间的事件。详情可参考套件 *Examples* 目录中提供的定制事件实例。此外，如果目标板是以 x86 或 PPC 为基础的系统，可以使用 IBM 提供的 DProbes 套件，在不需要重新编译的情况下为二进制文件（包括内核）加入跟踪点。DProbes 可以从 IBM 位于 <http://oss.software.ibm.com/developer/opensource/linux/projects/dprobes/> 的网站取得，它的发行采用的是 GPL 的许可条款。

LTT 可以从 Opersys 位于 <http://www.opersys.com/LTT/> 的网站取得，它的发行采用的是 GPL 的许可条款。该计划的网站还为 LTT 的用户提供了详细文件和邮件论坛的链接。目

前的稳定版为 0.9.5a，它支持的架构包括 i386、PPC 和 SH。目前的开发版为 0.9.6，它支持的架构还多了 MIPS 和 ARM。

事前准备

下载 LTT 套件并且它解开放到 `$(PRJROOT)/debug` 目录，接着移往套件的源码目录进行其余的操作：

```
$ cd $(PRJROOT)/debug
$ tar xvzf TraceToolkit-0.9.5a.tgz
$ cd $(PRJROOT)/debug/TraceToolkit-0.9.5
```

在套件中的 *Help* 目录里你可以看到跟计划网站一样的在线使用手册，该文件为 LTT 的使用提供了详细的说明。

修补内核

要让内核产生跟踪信息，必须修补内核。所有的内核补丁就放在 LTT 套件中的 *Patches* 目录里。然而，因为内核不断发展，所以经常需要更新内核补丁。你通常可以到 <http://www.opersys.com/ftp/pub/LTT/ExtraPatches/> 取得新版内核的补丁。以我的控制模块为例，我使用的是 *patch-ltt-linux-2.4.19-vanilla-020916-1.14*。如果使用的是不同的内核，可以试着将此补丁调整成符合内核版本。遗憾的是，每当有新的 LTT 版本发行时，我们很难提供一个适用所有内核版本的补丁。如果能够将补丁纳入主内核树，LTT 的使用将会变得简单许多。要说服内核开发者这么做，我还得下一番功夫。以我的控制模块为例，我必须修正经过修补的内核，因为进行修补时会出现无法修补的区段。

因为跟踪数据的二进制格式会不断发展变化，LTT 套件无法读取任意修补文件产生的数据。附加在补丁名称之后的 -1.14 标示出了此修补文件产生的跟踪数据的格式。LTT 0.9.5a 可以读取使用 1.14 格式的修补文件产生的数据。然而，它无法读取其他格式的数据。如果试图打开的跟踪数据的格式与可视化工具不兼容，可视化工具会显示错误信息并且离开执行状态。未来，LTT 的开发者准备修改相关工具以及跟踪数据的格式以避免此限制。

一旦下载你所选用的补丁之后，接着将它移往内核的源码目录，然后进行内核的修补：

```
$ mv patch-ltt-linux-2.4.19-vanilla-020916-1.14 \
> $(PRJROOT)/kernel/linux-2.4.18
$ cd $(PRJROOT)/kernel/linux-2.4.18
$ patch -p1 < patch-ltt-linux-2.4.19-vanilla-020916-1.14
```

现在你可以像前面那样配置内核。从配置主菜单进入“Kernel tracing”次级菜单并将“Kernel events tracing support”选项设成内建功能。在 LTT 0.9.6pre2 之前发行的补丁

中，例如我的控制模块使用的版本，还可以选择将跟踪功能设成模块，以动态方式加载跟踪驱动程序。然而，采用内核开发者的建议之后这个选项不见了，从此跟踪措施会被实现成一个内核子系统而不是一个设备。

接着你可以使用前几章提到的技术在目标板上建立以及安装内核。

尽管一旦系统开发好之后你可能会想让内核回到没有LTT的原始状态，不过我还是建议你保留这个可跟踪的内核，因为你无法确定实地应用时何时会出问题。我前面提到的“火星开拓者号”便属此例。最后JPL的工程师们以“test what you fly and fly what you test”的哲学解决了“开拓者号”的问题，详情参见前面在附注所提到的报导。请注意，当跟踪监控程序闲置时跟踪措施耗费的总体最大系统性能低于0.5%。

建立跟踪监控程序

正如前面所说，跟踪监控程序负责将跟踪数据写入永久性存储设备。尽管在大多数的工作站上这是个磁盘，不过使用通过NFS安装的文件系统来转储跟踪数据可能会比较好。当然你也可以将跟踪数据转储到目标板的MTD设备（如果有此设备的话），不过这么做必然会造成损耗增加的结果，因为跟踪数据一般都相当大。

回到套件的源码目录（位于`${PRJROOT}/debug`）建立跟踪监控程序：

```
$ cd ${PRJROOT}/debug/TraceToolkit-0.9.5
$ ./configure --prefix=${PREFIX}
$ make -C LibUserTrace CC=powerpc-linux-gcc UserTrace.o
$ make -C LibUserTrace CC=powerpc-linux-gcc LDFLAGS="-static"
$ make -C Daemon CC=powerpc-linux-gcc LDFLAGS="-static"
```

将LDFLAGS的值设成`-static`会产生以静态方式链接LibUserTrace链接库的二进制文件。这么做并不会对目标板造成太大的影响，因为这个链接库非常小。此外，这么做可以免除必须为目标板记住额外链接库的麻烦。尽管如此，我们产生的跟踪监控程序仍旧是以动态的方式链接C链接库。如果想要以静态的方式链接C链接库，可以改用如下的命令：

```
$ make -C Daemon CC=powerpc-linux-gcc LDFLAGS="-all-static"
```

所产生的二进制文件相当小。当跟踪监控程序以动态（或静态）方式链接glibc并经过strip处理，会产生18 KB（或350 KB）的二进制文件。当跟踪监控程序以动态（或静态）方式链接uClibc并经过strip处理，会产生16 KB（或37 KB）的二进制文件。

建立好之后，接着将跟踪监控程序以及若干基本的跟踪辅助命令脚本复制到目标板的根文件系统：


```
$ cp tracedaemon Scripts/trace Scripts/tracecore Scripts/traceu \  
> ${PRJROOT}/rootfs/usr/sbin
```

这些辅助命令脚本可以简化跟踪监控程序的使用,因为跟踪监控程序要能正常运行需要相当长的命令行。这些辅助命令脚本的用法可参考LTT文件的说明。我的经验是, *trace* 命令脚本是启动跟踪监控程序的最简单方法。

写作本书时,需要为跟踪设备在目标板的根文件系统中建立适当的设备条目,好让跟踪监控程序正确地与内核的跟踪组件交互。因为设备会在加载时取得它的主编号,所以请确定你在建立设备时使用的主编号是正确的。达成此目标的最简单方法就是先以正确的顺序加载所有的驱动程序并使用 *cat* 命令印出 */proc/devices* 文件的内容,接着你会取得一份设备主编号的清单,然后参考《Linux Device Drivers》(该书对主编号的分配有完整详细的说明)。或者,也可以使用LTT套件附带的 *createdev.sh* 命令脚本。以我的控制模块为例,分配给跟踪设备的主编号为254(注4)。

```
$ su -m  
Password:  
# mknod ${PRJROOT}/rootfs/dev/tracer c 254 0  
# mknod ${PRJROOT}/rootfs/dev/tracerU c 254 1  
# exit
```

正如前面所说,如果所使用的LTT版本比0.9.5a还新,可能不需要建立这些项目。详情参见套件附带的文件。

安装可视化工具

可视化工具运行在主机上,负责以直观方式显示跟踪数据。它可以当成命令行工具程序来用(以文字格式显示二进制的跟踪数据)或是当成GTK-based图形工具程序来用(将跟踪数据显示成跟踪图、一组统计数据,以及使用文字格式显示原始的跟踪数据)。图形接口无疑是分析跟踪数据的最简单方法,可是如果想要以命令脚本来分析文字格式的跟踪数据,可能会想要把它当成命令行工具程序来用。如果准备使用图形接口,系统上必须安装GTK。缺省情况下,大多数发行套件都会安装GTK。如果系统上并未安装GTK,此时可以使用发行套件的套件管理程序来安装它。

现在移往LTT套件的源码目录(前一节已经完成了配置设定的工作)建立以及安装主机组件:

```
$ make -C LibLTT install  
$ make -C Visualizer install
```

注4: 我取得此编号的方法是,加载跟踪驱动程序之后检查目标板上的 */proc/devices* 文件。

可视化工具的二进制文件 *tracevisualizer* 会被安装到 *\$(PREFIX)/bin* 目录、辅助命令脚本则会被安装到 *\$(PREFIX)/sbin* 目录。如同跟踪监控程序一样、辅助命令脚本可让你免除键入冗长的命令行来启动跟踪可视化工具的麻烦。

跟踪目标板以及将它的行为可视化

现在已经做好了跟踪目标板的准备。正如稍早所说、为了减少损耗应避免使用目标板的固态存储设备来记录跟踪数据。你可以改用经 NFS 安装的文件系统，或是如果不想让 NFS 造成的网络流量影响到跟踪数据，也可以先使用经 TMPFS 安装的目录来保存跟踪数据，等到完成跟踪后再将整个跟踪数据复制到主机。

下面这条简单的命令会跟踪目标板 30 秒的时间：

```
# trace 30 outt
```

此处所指定的 *outt* 将会被命令当成是输出文件的基本文件名。这条命令将会产生两种文件：

outt.trace（包含原始的二进制跟踪数据）以及 *outt.proc*（包含跟踪开始时系统状态的快照）。如果想在主机上使用可视化工具重建系统行为必须使用这两个文件。如果这两个文件在目标板上，请将它们复制到主机。

系统产生的事件很可能会超过跟踪措施可以处理的量。如果这样，监控程序将会在结束执行时通知你丢失了跟踪事件。于是你可以改变要使用的缓冲区的大小或者要跟踪的事件集，以便取得你需要的所有数据。欲了解“跟踪监控程序”接受哪些参数，可以参考套件附带的文件。

一旦包含跟踪数据的文件复制到主机之后，可以使用如下的命令行来检查跟踪数据：

```
$ traceview outt
```

这条命令会开启如图 11-1 所示的窗口。

图 11-1 显示了 BusyBox shell 与另一个 BusyBox 子进程的交互情况。你可以在可视化工具的左边窗格看到跟踪期间的所有活动进程。Linux 内核永远是这份进程清单的最后一个条目。你可以在可视化工具的右边窗格看到描述系统行为的图形。在该图形中、水平轴代表时间、垂直轴代表某个状态的转变。该图形显示系统在一开始执行的是内核的程序代码。启动之后不久，内核将控制权交给 *sh* 应用程序，由它执行一段时间直到进行 *wait4()* 系统调用。接着控制权会回到内核手上，由它执行一段时间直到调度器对 PID 编号为 21 的工作重新调度。接着该工作会开始执行，但是发生了一个例外事件，于是控制权再度回到内核手上。

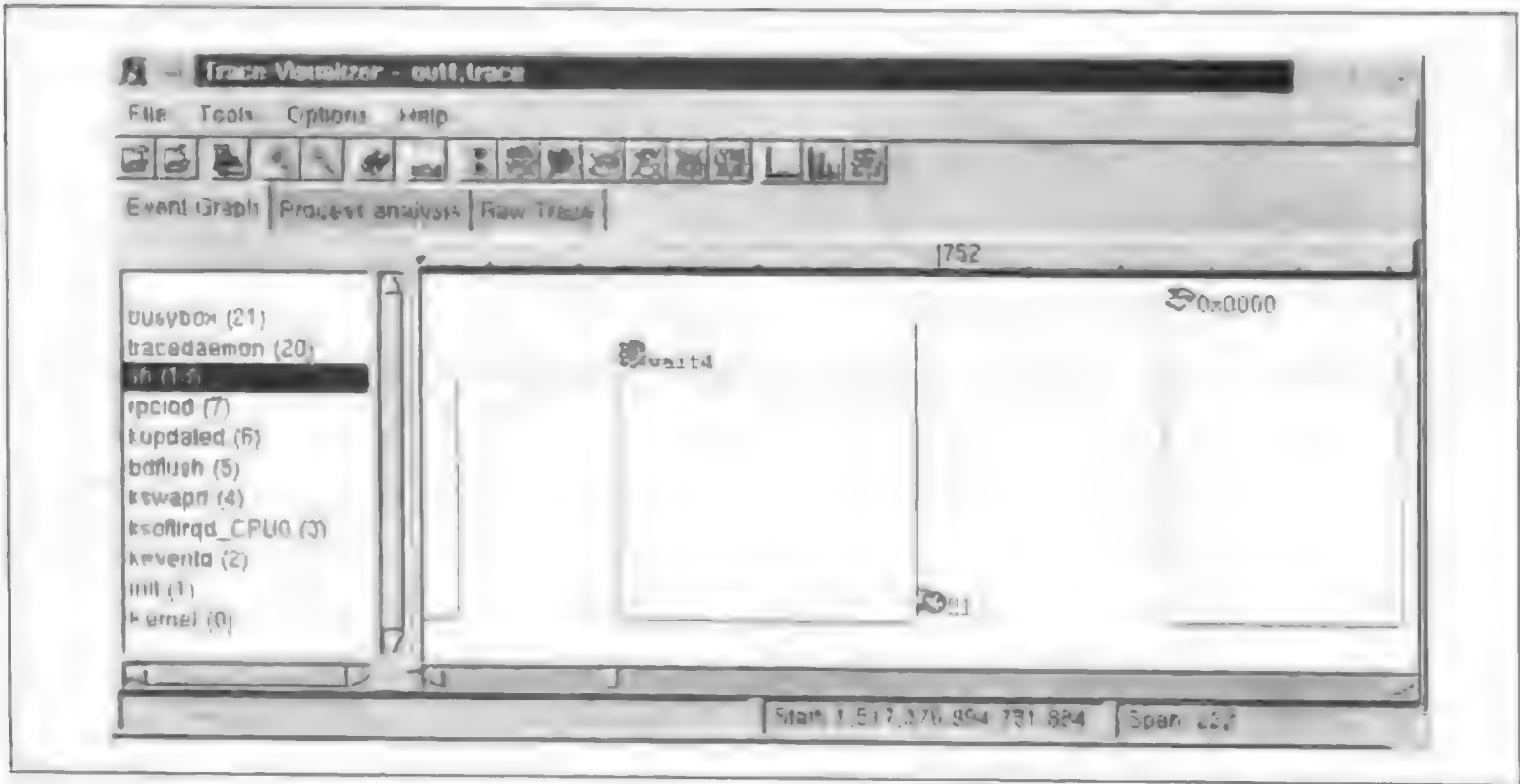


图 11-1: LTT 的跟踪图 (trace graph) 范例

此图形会向两个方向延伸,可以使用滚动条向左右移动来检查这个跟踪区段之前或之后发生了什么事。你也可以依照自己的需要将此图形放大或缩小。

使用这种图形,正如前面所说,可以轻易找出应用程序与系统中其他进程有哪些交互。你还可以使用可视化工具的“Raw Trace”功能以原始的形式来检查相同的一组事件,如图 11-2 所示。

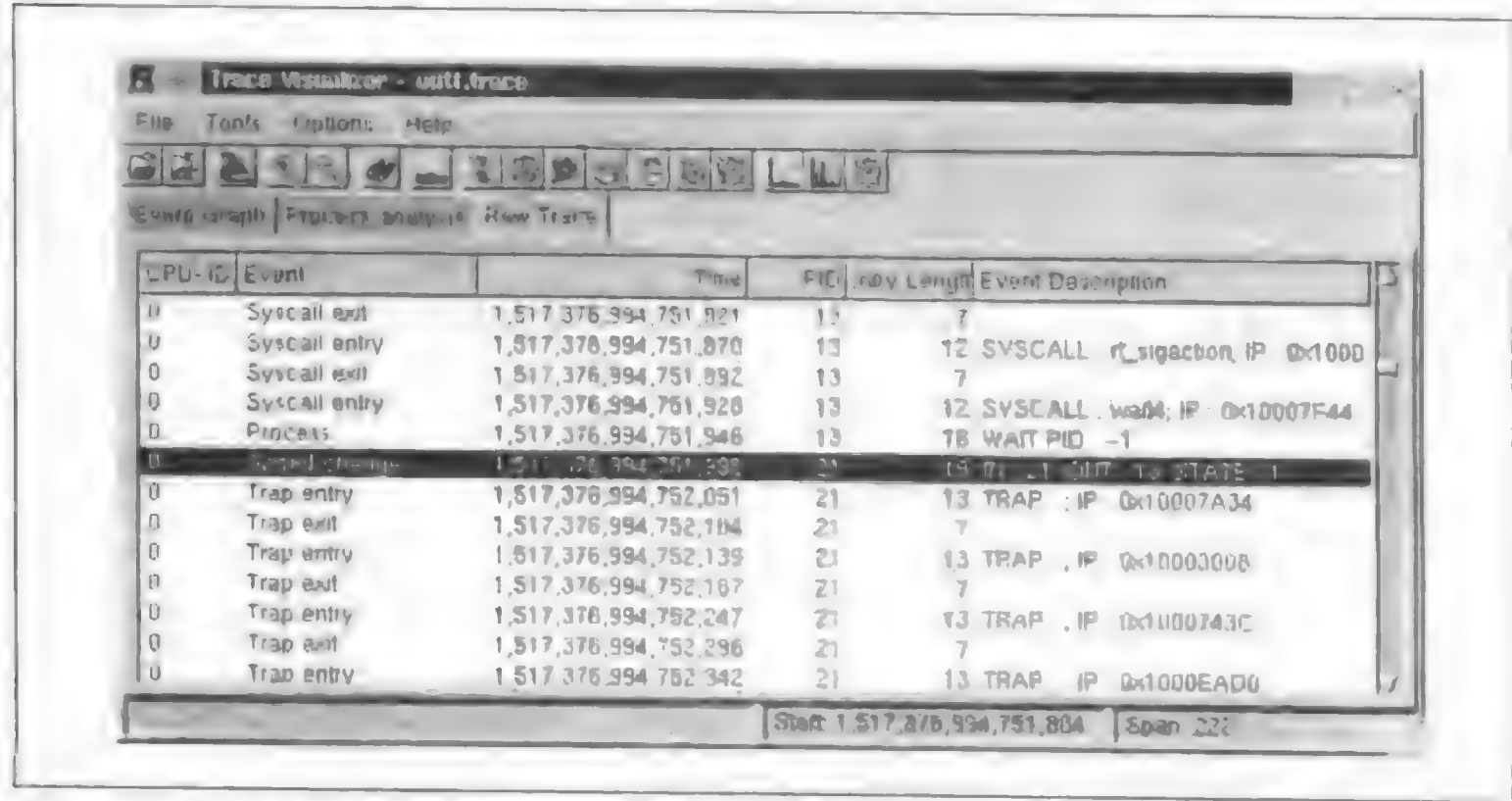


图 11-2: LTT 的原始事件清单 (raw event list) 范例

如果连图形工具都不想使用的话，可以把 *tracevisualizer* 当成是命令行工具来用。在这种用法中，*tracevisualizer* 命令会读取两个输入文件并且会产生一个内含原始事件清单的文字文件。这份清单的内容就是图形接口的“Raw Trace”功能显示的内容。要以文字的形式转储请键入：

```
$ tracevisualizer outt.trace outt.proc outt.data
```

这条命令的头两个参数 *outt.trace* 和 *outt.proc* 就是前面提到的那两个输入文件，跟踪数据的内容会以文字的形式转储到 *outt.data* 这个输出文件。你还可以使用 *tracedump* 或 *traceanalyzer* 之类容易使用的命令脚本来进行分析。本章稍后会探讨 LTT 的分析能力以及图形接口的“Process analysis”功能。

性能分析

全面取得与目标板性能有关的各种数据是能够善用目标板能力的关键所在。尽管此处的说明无法涵盖性能分析的所有方面，不过我会说明最重要的部分。接下来我们将会分节讨论进程统计、程序代码涵盖范围、系统统计、内核统计以及测量中断等待时间。

进程统计

进程统计是一个可以帮助你了解进程错综复杂行为的机制。此外，这还涉及了取得以下的信息：执行每个函数所花的时间，以及该时间有多少是花在替它的调用者工作，有多少是花在它所调用的子进程。

单一进程在 Linux 中的统计通常会使用一个特殊的编译器选项以及 *gprof* 工具程序。基本上，源码文件将这个编译器选项编译之后，会在运行时收集统计数据，并且会在应用程序结束时写入文件。然后你可以使用 *gprof* 分析这些资料，*gprof* 会产生调用图统计资料。尽管我并不打算说明 *gprof* 的实际用法以及解释它产生的输出（因为 GNU 的 *gprof* 使用手册已经有说明了），不过我将会说明它的跨平台用法的细节。

首先，必须修改应用程序的 Makefile，加入适当的编译器选项和链接器选项。下面是第四章提供的 Makefile 中，当你想要建立一个会产生统计数据的程序时，必须修改的部分：

```
CFLAGS      = -Wall -pg
...
LDFLAGS     = -pg
```

请注意，编译器标志和链接器标志都会包含 *-pg* 选项。*-pg* 编译器选项用来告诉编译器在所编译的源码中纳入产生性能数据的程序代码。*-pg* 链接器选项用来告诉链接器用

gcr1.o 而不是 *crt1.o* 来链接二进制文件。前者是（需要进行统计）后者的特殊版本。同时请注意，不要使用 *-O2* 这个编译器优化选项。这样可以确保所产生的应用程序会确实按照源码文件指定的方式进行。然后我们测量的才会是自己算法的性能，而不是经过编译器优化后的性能。

一旦重新编译好应用程序之后，接着将它复制到目标板加以执行。程序必须执行相当长的时间才会产生有意义的结果。请尽量为应用程序提供广泛的输入，让所有的程序代码尽可能被操作到。应用程序结束执行之后会产生一个内含统计数据的输出文件 *gmon.out*。此文件具有跨平台可读性，因此你可以使用主机的 *gprof* 来分析它。将 *gmon.out* 文件复制到相应应用程序的源码目录之后，接着使用 *gprof* 取出调用图统计数据：

```
$ gprof command-daemon
```

这条命令会从标准输出印出调用图统计数据。如果需要的话，可以使用 *>* 运算符将此输出重定向到一个文件。你并不需要特别指定 *gmon.out* 文件，因为它会被自动加载。关于 *gprof* 用法的更多信息请参考 GNU 的 *gprof* 使用手册。

程序代码涵盖范围

除了确认应用程序各个部分所耗费的时间，计算应用程序中每条语句执行多少次也是有意义的。这种涵盖范围分析可以揭露哪些程序代码从未被调用或是哪些程序代码经常被调用，应该予以特别注意。

进行涵盖范围分析最常见的方法就是使用编译器选项以及 *gcov* 工具程序。该功能是在编译 *gcc* 编译器的时候一起被编译进 *gcc* 链接库 *libgcc* 的。

然而，不幸的是，当 *gcc* 在 3.0 之前的版本侦测到其建立的是交叉编译器的时候，便不许允将涵盖范围分析的功能编译进 *libgcc*。例如，对第四章所建立的编译器来说，其 *libgcc* 就不会包含可以产生程序代码涵盖范围相关数据的适当程序代码。因此，除非修改 *gcc* 的源码，否则无法进行程序的涵盖范围分析。

要使用 *gcc* 3.0 以后的版本来建立需要涵盖分析的程序代码，只须在设定它们的“建立配置”的时候使用 *--with-headers=* 选项即可。

欲避免 *gcc* 版本在 3.0 之前的问题，请编辑 *gcc-2.95.3/gcc/libgcc2.c* 文件（或是在编译器版本中等效的文件），禁用如下的定义：

```
/* In a cross-compilation situation, default to inhibiting compilation
   of routines that use libc.  */
```

```

#if defined(CROSS_COMPILE) && !defined(inhibit_libc)
#define inhibit_libc
#endif

```

要禁用此定义，请在这段程序代码前后分别加上 `#if 0` 和 `#endif`，结果如下所示：

```

/* gcc makes the assumption that we don't have glibc for the target,
   which is wrong in the case of embedded Linux. */
#if 0

/* In a cross-compilation situation, default to inhibiting compilation
   of routines that use libc.  */

#if defined(CROSS_COMPILE) && !defined(inhibit_libc)
#define inhibit_libc
#endif

#endif /* #if 0 */

```

现在重新编译并重新安装 `gcc`，就像我们在第四章所做的那样。但是你别建立引导编译器，因为我们已经建立并安装好了 `glibc`。我们只须建立完整编译器。

接下来修改应用程序的 `Makefile`，让它使用适当的编译器选项。下面是第四章提供的 `Makefile` 中，当你想要建立一个会产生“程序代码涵盖范围”数据的程序时，必须修改的部分：

```

CFLAGS      = -Wall -fprofile-arcs -ftest-coverage

```

正如前面在编译会产生统计数据的应用程序时提到的，必须去掉 `-O` 优化选项，这样你才会取得真正与原始程序代码相对应的“程序代码涵盖范围”数据。

源码目录中每个经过编译的源码文件现在应该都会具备两个文件（`.bb` 和 `.bbg`）。请将程序复制到目标板，并以正常的方式执行它。当你执进程序时，将会为每个源码文件产生一个 `.da` 文件。然而，不幸的是，这个 `.da` 文件是使用最初源码文件的绝对路径产生的。因此，必须在目标板的根文件系统上为此路径建立一个副本。尽管你可能无法从该目录执行二进制文件，不过应用程序将会把 `.da` 文件放到该处。以我的“命令监控程序”为例，它就位于主机上的 `/home/karim/control-project/control-module/project/command-daemon` 目录。因此我必须在目标板的根文件系统上建立完整的路径，让监控程序的 `.da` 文件能够正确产生。`mkdir` 的 `-p` 选项在此例中相当有用。

一旦程序执行完成之后，接着将 `.da` 文件复制回主机，然后执行 `gcov`：

```

$ gcov daemon.c
71.08% of 837 source lines executed in file daemon.c
Creating daemon.c.gcov.

```

这条命令产生的 *.gcov* 文件会包含可供人读取的“涵盖范围”信息。*.da* 文件与架构无关，所以使用主机的 *gcov* 来处理它一点问题都没有。欲取得关于 *gcov* 使用或产生的输出的更多信息，可参考 *gcc* 使用手册的“*gcov*”这一节。

系统统计

每个 Linux 系统都会存在许多竞争系统资源的进程。当你想要建立一个负载均衡而且有良好的响应性能的系统时，能够把每个进程对系统的负荷所造成的影响量化很重要。Linux 中有若干基本的方法可用来把进程对系统的影响量化。这一节将会探讨其中两种方法：从 */proc* 取出信息以及使用 LTT。

关于 */proc*

内核会在 */proc* 文件系统包含的虚拟条目中提供与其内部数据结构和系统有关的信息。这些信息有部分（例如进程时间）是内核在每个时钟周期取样收集而成的。要从 */proc* 目录取出信息，传统上会使用 *procps* 套件，此套件包含 *ps* 和 *top* 之类的工具程序。目前有两个独立维护的 *procps* 套件。第一个套件由 Rik van Riel 维护，可以从 <http://surriel.com/procps/> 取得。第二个套件由 Albert Cahalan 维护，可以从 <http://procps.sourceforge.net/> 取得。尽管关于哪个是正式的 *procps* 套件争议不断，不过这两个套件包含的 *Makefile* 都不适合跨平台开发，因此也不适合在嵌入式系统中使用。你可以改用 *BusyBox* 提供的 *ps* 替代品。尽管它不会像 *procps* 的 *ps* 那样输出进程统计数据，不过它会提供与目标板上所执行软件有关的基本信息：

```
# ps
  PID  Uid        VmSize Stat Command
    1  0             820 S    init
    2  0             S    [keventd]
    3  0             S    {kswapd}
    4  0             S    {kreclaimd}
    5  0             S    [bdflush]
    6  0             S    [kupdated]
    7  0             S    [mtdblockd]
    8  0             S    [rpciod]
   16  0             816 S    -sh
   17  0             816 R    ps aux
```

如果觉得以上的信息仍不够用，还可以手动浏览 */proc* 中每个进程来取得需要的信息。

使用 LTT 进行完整的统计

因为 LTT 会记录重要的系统信息，所以它可以取出关于系统行为极详细的信息。与 */proc* 目录中找到的信息不同，LTT 产生的统计资料并非取样来的，它依据的是内核内部针对

每个进程耗费时间精确记帐的结果，LTT提供了两种统计资料：针对每个进程的统计数据以及针对系统的统计数据。在图形接口的“Process analysis”功能中都提供了这两种统计数据。

当你在图形接口的“Process analysis”功能中显示的进程树里选择一个进程，LTT便会针对该进程显示统计资料。图 11-3 展示了你可以对单一进程取出哪些数据。

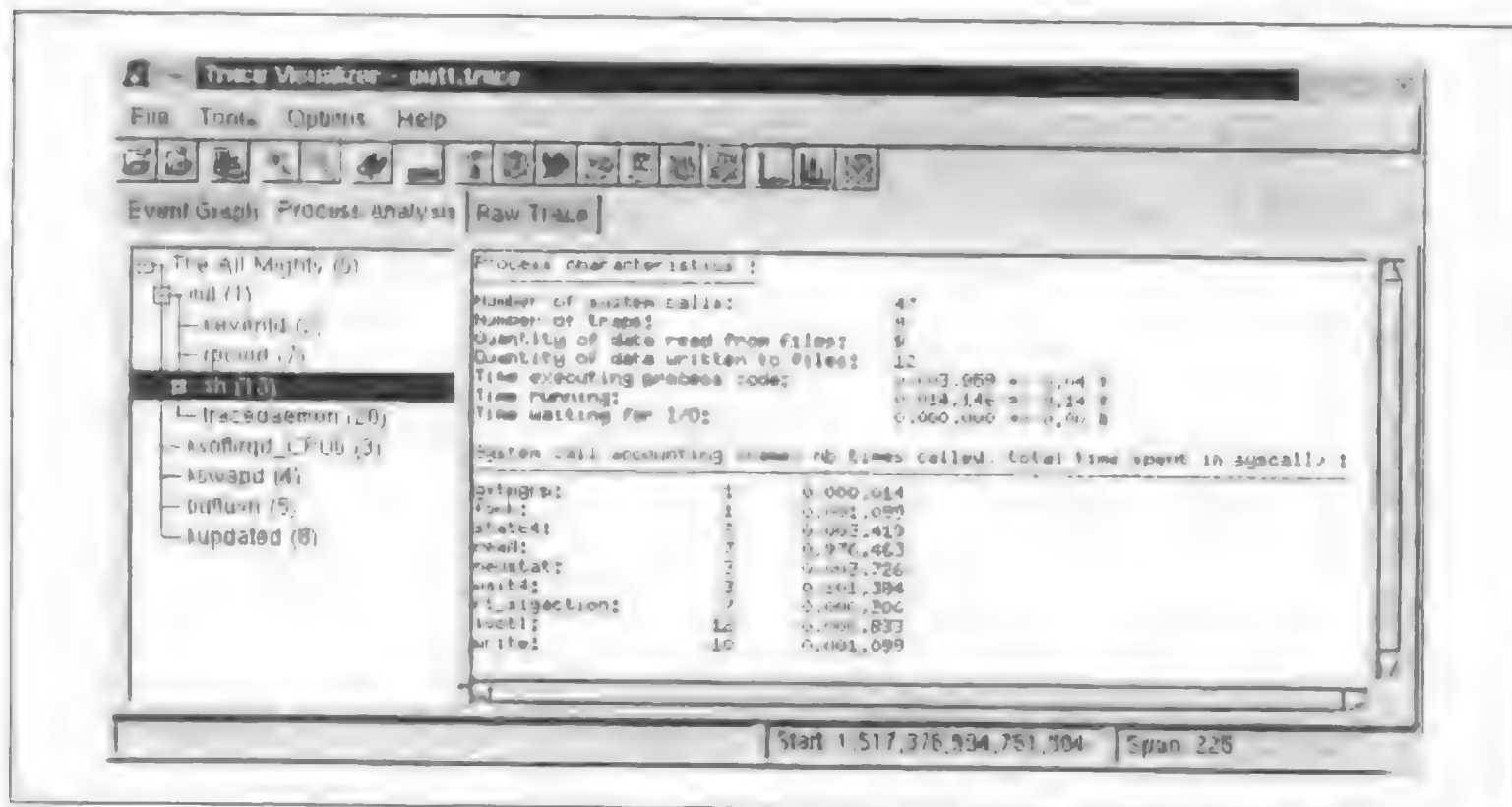


图 11-3: 单一进程的统计资料

此外，这些数据可以告诉你，选定的工作被内核运行了多少时间以及实际执行进程代码耗费的时间。此例中，选定的工作并未等待任何的 I/O。但是如果它有等待 I/O，“Time waiting for I/O”那一行会提供等待 I/O 耗费时间的测量值。此处提供的时间和百分比取决于跟踪所花的时间。此例中，跟踪了 10 秒的时间。

LTT 还会提供与应用程序用到的系统调用有关的信息。尤其是，它还提供了每个系统调用被调用的次数，以及内核为这些调用提供服务花费的时间。

当你选择进程树中最顶层的进程条目（The All Mighty (0)），LTT 会显示整个系统的统计数据。图 11-4 展示了 LTT 取出的系统数据。

该系统开头的数字是关于跟踪本身的统计数据。此例中，差不多跟踪了 10 秒的时间，而且其中大约有 98% 的时间系统是闲置的，接着会提供若干关键事件发生的次数。在 7467 笔事件中，LTT 表示有 1180 笔 trap 事件以及 96 笔中断事件（有 96 笔 IRQ 进入和 96 笔 IRQ 离开）。这类信息可以协助你找出系统整体行为的实际问题。此画面还会显示各应用程序用到的系统调用的累计值。

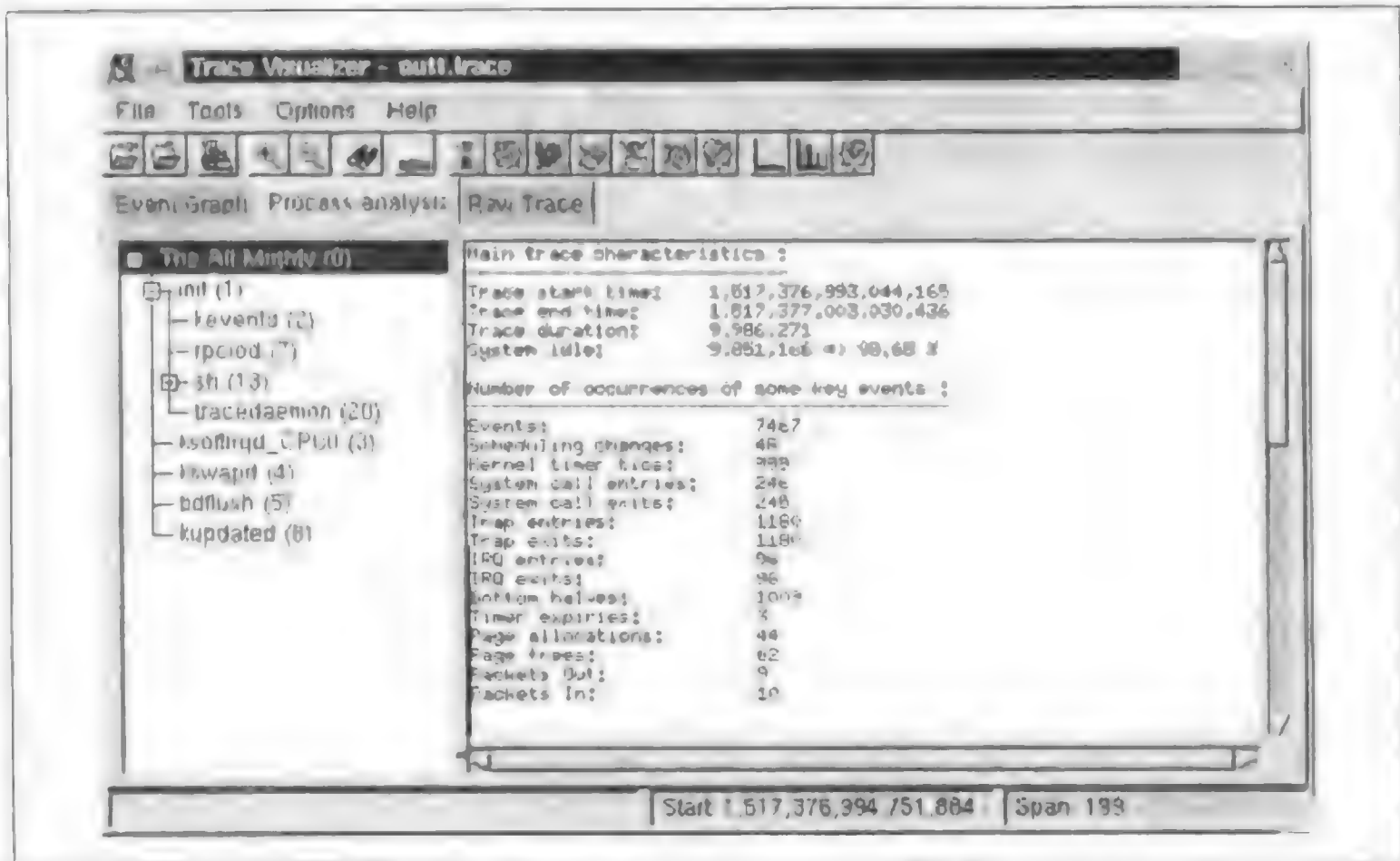


图 11-4: 整个系统的统计资料

如同实际的跟踪信息，图形接口的“Process analysis”功能中显示的统计资料也可以从命令行以文字的形式转储到文件。欲取得如何完成此事的更多信息，请参考LTT的说明文件。

内核统计

有时应用程序并非是性能降低的根源，它可能受到内核本身性能不佳的影响。在这种情况下，需要使用正确的工具来找出内核之所以会这样的原因。

目前可以测量内核性能的工具还真不少。其中最最著名的可能是LMbench (<http://www.bitmover.com/lmbench/>)。然而，LMbench需要用到C编译器和Perl解释器，因此不适合嵌入式系统使用。另一个可以测量内核性能的工具是kernprof (<http://oss.xgt.com/projects/kernprof/>)。尽管它产生的输出可供gprof读取，不过它的安装涉及到了内核的修补而且只能使用在x86、ia64、sparc64和mips64架构上。正如所见，大多数的嵌入式架构kernprof几乎都不支持。

事实上，内核内建了以取样为基础的统计功能。这个统计系统的工作方式就是在每次定时器中断的时候取样指令指针。经过长期的取样之后，估计内核花最多时间执行的函数其命中次数应该会高于其他函数。尽管这是一个粗糙的统计方法，不过此刻对大多数嵌入式Linux系统来说算是最合适的统计方法。

要启用内核的统计功能，必须使用 `profile=` 引导参数。你提供的参数值是用来设定、在指令指针作为取样表的索引之前应该右移的位数。此数值越小取样的精确度就越高，不过取样表就需要越多的内存空间。此值通常会被设成 2。

取样活动本身并不会使系统变慢，因为它只会发生在每个时钟周期，而且计数器的递增可以在定时器中断时通过指令指针的值轻易完成。

一旦你在内核启动时传递了 `profile=` 参数给它，将会在目标板的 `/proc` 目录中找到一个新的条目 `/proc/profile`，这就是内核输出的取样表。

要从 `/proc/profile` 读出取样数据，必须使用 `readprofile` 工具程序；`readprofile` 有两种形式：一个独立的套件（可以从 <http://sourceforge.net/projects/minilop/> 取得）或是 `util-linux` 套件（可以从 <http://www.kernel.org/pub/linux/utils/util-linux/> 取得）的一个组件。接下来我将只会探讨独立套件形式的 `readprofile`，因为 `util-linux` 所包含的不只是 `readprofile`。下载 `readprofile` 套件并将它解开放到 `${PRJROOT}/debug` 目录。接着移往套件的源码目录并且编译工具程序：

```
$ cd ${PRJROOT}/debug/readprofile-3.0
$ make CC=powerpc-uclibc-gcc
```

欲以静态方式来编译工具程序，请在 `make` 命令行上加入 `LDFLAGS="-static"`。其产生的二进制文件相当小。举例来说，当它以静态方式链接 `uClibc` 并且经过 `strip` 处理会产生 30 KB 的二进制文件。

一旦 `readprofile` 建立好之后，接着将它复制到目标板的 `/usr/bin` 目录：

```
$ cp readprofile ${PRJROOT}/rootfs/usr/bin
```

要让 `readprofile` 运行正常，还必须将适当的内核映射文件 `System.map` 复制到目标板的根文件系统：

```
$ cp ${PRJROOT}/images/System.map-2.4.18 ${PRJROOT}/rootfs/etc
```

目标板根文件系统准备好之后，修改内核引导参数并且加上 `profile=2` 引导参数。系统引导之后，可以执行 `readprofile`：

```
# readprofile -m /etc/System.map-2.4.18 > profile.out
```

`profile.out` 文件现在包含了文字形式的统计信息。你可以在任何时候以“对 `/proc/profile` 进行写入操作”的方式（注 5）来抹除目标板上收集的取样表：

```
# echo > /proc/profile
```

注 5： 你并不需要写入任何内容。这只是一个抹除统计信息的写入动作而已。

完成统计之后，接着将 *profile.out* 文件复制回主机并检查它的内容：

```
$ cat profile.out
...
30 __save_flags_ptr_end      0.3000
10 __sti                     0.1250
8 __flush_page_to_ram       0.1053
7 clear_page                 0.1750
3 copy_page                  0.0500
1 m8xx_mask_and_ack         0.0179
2 iopa                       0.0263
1 map_page                   0.0089
...
1 do_xprt_transmit           0.0010
1 rpc_add_wait_queue         0.0035
1 __rpc_sleep_on             0.0016
1 rpc_wake_up_next           0.0068
1 __rpc_execute              0.0013
2 rpciod_down                0.0043
15 exit_devpts_fs            0.2885
73678 total                  0.0618 0.04%
```

最左边的字段代表相应位置被取样的次数，第二字段代表被取样的函数，第三个字段提供的是函数负荷的近似值（函数的使用周期和函数长度的比值）。欲深入了解 *readprofile* 之输出的细节，请参阅此套件附带的 *manpage*。

测量中断等待时间

对实时嵌入式系统来说一个最重要的度量值就是它们响应外界事件所花的时间。这类系统，正如第一章所说，如果未及时响应外界事件，可能会造成灾难。已知有若干特别的技术可用来测量系统对中断的响应时间（常被称为中断等待时间）。这些测量技术大致可以分成两类：

自给自足式

由系统自己触发中断。欲使用此技术，必须将系统的一个输出管脚连接到一个“中断产生”输入管脚。以 PC-based 的系统为例，可以将适当的并行端口管脚连在一起，详情参见《Linux Device Drivers》。对其他类型的系统来说，这可能涉及了较复杂的设置。

诱发式

此技术用外部的信号源（例如频率发生器）来触发中断。你可以将这个信号源连接到目标板的“中断产生”输入管脚。

欲采用“自给自足测量法”，必须撰写一个小型的软件驱动程序来进行中断的初始化和处理。欲进行中断的初始化，驱动程序必须做两件事：

1. 记录目前的时间。取得时间的方式通常会使用内核函数 `do_gettimeofday()`，这个内核函数可以提供微秒的分辨率。如果想取得更精确的时间值，还可以使用 `get_cycles()` 函数读取机器的硬件周期。举例来说，在 Pentium 等级的 x86 系统上，此函数会传回 TSC 寄存器的内容。然而，在 ARM 之上，此函数总是传回 0。
2. 切换触发中断的输出位。以 PC-based 的系统为例，这只是将适当的字节值写入并行端口的数据寄存器。

另一方面，此驱动程序的中断处理例程必须做两件事：

1. 记录目前的时间。
2. 切换输出管脚的位值。

将调用“中断处理例程”的时间点减去触发中断的时间点，就可以算出与实际非常接近的中断等待时间。这样之所以无法算出实际的“中断等待时间”，是因为你测量的时间包含了 `do_gettimeofday()` 和其他软件的执行时间。所以你必须让驱动程序重复运行多次以便计算“中断等待时间”的差异。

使用“自给自足测量法”时，要取得较好的“中断等待时间”测量值，请在驱动程序将会进行位值切换的输出管脚接上示波器观察位值切换所耗费的时间。此时间值应该略小于使用 `do_gettimeofday()` 取得的时间，因为示波器的输出不包含首次调用此函数的执行时间。要取得更好的“中断等待时间”测量值，必须将 `do_gettimeofday()` 的调用完全移除，并且测量位值前后两次切换所耗费的时间。

“自给自足测量法”适合在同时具备触发中断和处理中断功能的系统上使用，“诱发测量法”则通常是测量“中断等待时间”最可信的方法，也是最接近实际给系统传递中断信号的方法。举例来说，如果有个驱动程序具有较长的“等待时间”并且包含变更中断屏蔽的程序代码，那么“自给自足测量法”的中断驱动程序有办法触发中断之前，可能必须等到“等待时间”较长的驱动程序完成。然而“诱发测量法”并不会因此失败，因为中断的触发源与所测量的系统并无依存关系。

为“诱发测量法”撰写软件驱动程序比“自给自足测量法”简单得多。基本上，驱动程序必须实现一个“中断处理例程”来切换系统某个输出管脚的状态。通过同时描绘出系统的响应以及频率发生器产生的方波，可以精确测量出系统响应中断花费的时间。除了示波器，还可以使用简单的计数器电路来计算“中断触发”信号与“目标板响应”信号的时间差。“中断触发”信号会重置计数器电路，当此电路收到“目标板响应”信号时便会停止计数。你还可以使用另一个专门测量“中断触发”信号与“目标板响应”信号时间差的系统。

“自给自足测量法”和“诱发测量法”或它们的任何变体可能都很有效，然而Linux并非实时操作系统。因此，尽管当系统闲置时你可以看到稳定的“中断等待时间”，不过当系统的处理负荷增加时Linux的响应时间可能会变得非常大。进行“中断等待时间”测试时，可以试着在目标板上键入`/s-R/`来增加目标板的处理负荷，并且检查闪烁的示波器输出，观察这会造成什么影响。

你可能会想在系统负荷最大的时候测量“中断等待时间”。这将会在目标板上产生最大的“中断等待时间”。然而，应用程序可能不接受这个“等待时间”。如果需要取得最小的“中断等待时间”，可以考虑使用第一章提到的实时系统。

内存调试

与桌上型Linux系统不同的是，嵌入式Linux系统不能让应用程序因为非法内存参用产生转储数据而耗尽内存空间。此外，并没有任何用户会停止引起问题的应用程序并去重新启动它们。为嵌入式Linux系统开发应用程序期间，可以使用特殊的调试链接库以确保它们能够正确地使用内存空间。接下来我们会分节讨论这种类型的两个链接库：Electric Fence和MEMWATCH。

尽管这两个链接库都值得你在开发期间链接到应用程序，不过成为产品之后就不应该再链接这两链接库中的任何一个。首先，这两个链接库会以自己（对调试而非性能做过优化）的版本来替换C链接库的内存配置函数。其次，这两个链接库的发行采用的是GPL的许可条款。因此，尽管你可以在公司内部使用MEMWATCH和Electric Fence来测试应用程序，但如果不想以GPL许可条款来发行应用程序，就不可以在对外界发行应用程序时纳入这两个链接库。

Electric Fence

Electric Fence链接库为C链接库的内存配置函数（例如`malloc()`和`free()`）实现了具备有限测试功能的等效函数。因此，它能够非常有效地侦测越界内存参用。基本上，链接Electric Fence链接库之后，当有任何越界参用的状态发生时，都将会使得应用程序执行失败并进行内存转储。你可以通过在`gdb`中执行应用程序，立即找到有问题的指令。

Electric Fence的撰写者和维护者是Bruce Perens。你可以到<http://perens.com/FreeSoftware/>取得此套件。首先下载套件并且将它解开放到`$(PRJROOT)/debug`目录。以我的控制模块为例，我使用的是Electric Fence 2.1。

接着移往套件的源码目录以便进行其余操作：

```
$ cd ${PRJROOT}/debug/ElectricFence-2.1
```

为目标板编译 Electric Fence 之前，必须编辑 *page.c* 源码文件，并且将下面这段程序代码注释掉，也就是在这段程序代码前后分别加上 `#if 0` 和 `#endif`：

```
#if ( !defined(sgi) && !defined(_AIX) )
extern int      sys_nerr;
extern char *    sys_errlist[];
#endif
```

如果不用以上方式修改这段程序代码，Electric Fence 将会编译失败。把这段程序代码修改好之后，接着为目标板编译和安装 Electric Fence：

```
$ make CC=powerpc-linux-gcc AR=powerpc-linux-ar
$ make LIB_INSTALL_DIR=${TARGET_PREFIX}/lib \
> MAN_INSTALL_DIR=${TARGET_PREFIX}/man install
```

Electric Fence 链接库 *libefence.a*（内含内存分配函数的替代品）现在已经被安装到 *\${TARGET_PREFIX}/lib*。若想使用 Electric Fence 链接应用程序，必须为链接器的命令行加上 *-lefence* 选项。下面是我对“命令监控程序”的 Makefile 所做的修改：

```
CFLAGS      = -g -Wall
...
LDFLAGS     += -lefence
```

如果想让 *gdb* 印出引发问题的那一行程序代码，必须使用 *-g* 选项。编译并且经过 *strip* 处理之后，Electric Fence 链接库会让二进制文件增加大约 30 KB。建立好之后，像往常那样将二进制文件复制到目标板。

在目标板上执行此程序将会获得如下的结果：

```
# command-daemon

Electric Fence 2.0.5 Copyright (C) 1987-1998 Bruce Perens.
Segmentation fault (core dumped)
```

由于无法将 *core* 文件复制回主机进行分析，因为它由不同架构的系统产生，可以先在目标板上启动 *gdb* 服务器，然后使用目标板 *gdb* 从主机连到该服务器。下面是我在目标板上启动命令监控程序进行 Electric Fence 调试的例子：

```
# gdbserver 192.168.172.50:2345 command-daemon
```

在主机上我会这么做：

```
$ powerpc-linux-gcc command-daemon
(gdb) target remote 192.168.172.10:2345
Remote debugging using 192.168.172.10:2345
```

```
0x10000074 in _start ()
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x10000384 in main (argc=2, argv=0x7ffff794) at daemon.c:126
126 input_buf[input_index] = value_read;
```

此例中，引发非法内存参用的原因是 *daemon.c* 文件中第 126 行程序代码对数组进行了越界写入的操作。欲取得更多关于如何使用 Electric Fence 的信息，可参考该套件附带的大量 manpage。

MEMWATCH

MEMWATCH 链接库为一般的内存分配函数（例如 `malloc()` 和 `free()`）实现了可以记录分配状态的等效函数。它可以非常有效地侦测到内存泄漏错误，例如你忘了释放某个内存区段或是你重复释放了每个内存区段。这一点对嵌入式系统尤其重要，因为并没有人监控该设备，随时检查各种应用程序是否用完所有的内存空间。然而，就检测错误指针来说，MEMWATCH 并不如 Electric Fence 有用。例如，MEMWATCH 无法检测前一节提到的有问题的数组写入操作。

MEMWATCH 可以从它的计划网站 (<http://www.linkdata.se/sourcecode.html>) 取得。请下载该套件并将它解压到 `$(PRJROOT)/debug` 目录。MEMWATCH 由一个头文件和一个 C 程序文件组成，它们必须与应用程序编译在一起。要使用 MEMWATCH，请将这两个文件复制到应用程序的源码目录：

```
$ cd $(PRJROOT)/debug/memwatch-2.69
$ cp memwatch.c memwatch.h $(PRJROOT)/project/command-daemon
```

修改 Makefile，将新的 C 程序文件加入欲编译和链接的目标文件清单中。以我的“命令监控程序”为例，我会把 Makefile 修改成：

```
CFLAGS      = -O2 -Wall -DMEMWATCH -DMW_STUDIO
...
OBJS        = daemon.o memwatch.o
```

你还必须在源码文件中加入引用 MEMWATCH 头文件的语句：

```
#ifdef MEMWATCH
#include "memwatch.h"
#endif /* #ifdef MEMWATCH */
```

你现在可以像往常那样进行交叉编译。MEMWATCH 的使用并不需要什么特别的安装步骤。应用程序一旦建立好并经过 `strip` 处理之后，*memwatch.c* 和 *memwatch.h* 会让二进制文件的大小增加 30 KB。

当程序执行时，它会针对程序的行为产生一份报告，这份报告会放在二进制文件执行时所在目录的 *memwatch.log* 文件中。以下的内容是从我的“命令监控程序”执行时产生的 *memwatch.log* 文件中摘录的：

```
..... MEMWATCH 2.69 Copyright (C) 1992-1999 Johan Lindh .....  
...  
unfreed: <3> daemon.c(220), 60 bytes at 0x10023fe4      {FE FE FE ...  
...  
Memory usage statistics (global):  
  N)umber of allocations made: 12  
  L)argest memory usage      : 1600  
  T)otal of all alloc() calls: 4570  
  U)nfreed bytes totals      : 60
```

unfreed: 这一行信息会告诉你源码中哪一行分配的内存空间后来没有释放。此例中，*daemon.c* 的第 220 行分配的 60 bytes 没有释放。Total of all alloc() calls: 这一行信息会告诉你程序执行期间总共分配了多少内存空间。此例中，程序总共配置了 4570 字节的内存空间。

欲进一步了解如何使用 MEMWATCH 及其提供的输出，请参考套件附带的 *FAQ*、*README* 和 *USING* 文件。

关于硬件工具

本章的内容将重点放在用来对嵌入式 Linux 软件进行调试的软件工具上。除此之外，事实上还有许多硬件工具和辅助工具可以用来进行嵌入式软件的调试。正如本章前面所说，目标板上不同的操作系统使用硬件调试工具的方式各有一些的差异。尽管进行软件调试时硬件工具有时会比软件工具有效率，但是硬件工具的一个缺点就是它们的价格昂贵。举例来说，一台令人满意的 100 MHz 示波器其价格高达一千美元。尽管如此，还是让我们来看一下你可能会用来对运行 Linux 的嵌入式目标板进行调试的若干硬件工具。

注意： 尽管新的硬件工具似乎都很贵，不过租用或购买二手硬件工具或许可以为你省下不少钱。有些公司实际上就是专门提供出租和翻新硬件工具的业务。

能够协助你进行开发工作的最基本工具应该非示波器莫属了。正如我们在前文所见，它可以用来测量“中断等待时间”。然而，它还有许多其他的用途，例如：观察目标板与外界的交互以及监控电路板的内部信号。

尽管用示波器来监控极少数的信号相当有效，不过它并不适合用来分析有许多信号线同时进行传输的状态，例如系统的内存或 I/O 总线。欲分析此类流量，必须使用逻辑分析

仪。它可以让你检查通过总线传送的各种值。以地址总线为例，逻辑分析仪将可以让你看到地址线传送的实际地址。此工具还可以让你区分干扰和异常信号。

如果遇到非信号层次的问题，而是由于操作系统软件的缺陷或不成熟，那么你可能需要使用在线仿真器（In-Circuit Emulator, ICE）或是 BDM 或 JTAG 调试器。前者依靠的是拦截处理器和系统其他组件的交互，后者依靠的是将调试功能实现在处理器的硅芯片上并以若干特殊的管脚提供此功能（参见第二章的说明）。因为许多理由，ICE 有逐渐被 BDM 或 JTAG 调试器取代的趋势。然而，它们都可让你使用硬件强制能力来对操作系统内核进行调试。举例来说，可以使用此类工具来对崩溃的 Linux 内核进行调试。事实上，Linux 内核通常是在 BDM 和 JTAG 调试器的协助下移植到新架构的。如果打算从头开始建立自己的嵌入式系统，应该考虑为开发者提供 BDM 或 JTAG 接口，这样他们才能够附接 BDM 或 JTAG 调试器，即使它的价钱可能很高。大多数的商业嵌入式电路板都已经配备适当的连接器。

目前至少有一个开放源码的 BDM 调试器具备有完整的 *gdb* 补丁和硬件示意图。此计划称为 BDM4GDB，它的网站位于 <http://bdm4gdb.sourceforge.net/>。然而，此计划仅支持 MPC 860、850 和 823 PowerPC 处理器。尽管就其本身而言成效卓著，不过 BDM4GDB 并非是一个通用的 BDM 调试器。

LART 计划（<http://www.lart.tudelft.nl/>）为它的 StrongARM-based 系统提供了一个用来编程 flash 的 JTAG 转换器。这个转换器的示意图以及需要的软件都可以从 <http://www.lart.tudelft.nl/projects/jtag/> 取得。这个转换器只能用来重新编程 flash 设备，它无法用来进行系统调试。因此，仍然需要一个真正的 JTAG 调试器。

如果对“使用硬件工具对嵌入式系统进行调试”这个主题不熟悉，建议你参考 Arnold Berger 的著作《Embedded Systems Design》（CMP Books）以及 Jack Ganssle 的著作《The Art of Designing Embedded Systems》（Newnes）。如果目前从事设计或修改目标硬件的工作，可能会对 John Catsoulis 的著作《Designing Embedded Hardware》（O'Reilly）感兴趣。

附录一

工作单

尽管不同的嵌入式 Linux 系统差异相当大，但是本书描述的方法应该可以立即用来建立任何类型的嵌入式 Linux 系统。之所以能够这样，是因为本书安排了一组规则，这组规则可以明确指出任何嵌入式 Linux 系统的特性。本附录提供的工作单就是这组规则。开发完成之后，任何开发者只要使用该工作单并搭配本书的说明，不需要原设计者的协助就能重建嵌入式 Linux 系统。开发期间，开发小组的成员可以使用此工作单取得系统每个组件的详细信息。

此工作单为嵌入式 Linux 系统的每个层面提供了一节来描述其细节。每一节都由与“相应嵌入式 Linux 系统层面”有关的属性组成。这些节是：

- 项目标识符 (Project identification)
- 硬件摘要 (Hardware summary)
- 开发工具 (Development tools)
- 内核 (Kernel)
- 根文件系统 (Root filesystem)
- 存储设备结构 (Storage device organization)
- 引导加载程序的配置与使用 (Bootloader configuration and use)
- 网络服务 (Networking services)
- 定制项目软件 (Custom project software)
- 调试记录 (Debug notes)
- 附注 (Additional notes)

大部分的节都会包含“Main contact”字段。此字段应该用来指定开发期间相应嵌入式系统的负责人名称。特定系统的负责人应该知道各种注意事项，以及掌握相关开放源码

和自由软件套件的最新发展动态。举例来说，内核负责人应该订阅Linux内核的邮件论坛以及与系统使用的架构有关的内核开发论坛。

虽然此工作单已经尽可能列出所有条目，不过你可能还需要根据项目的目的对它进行修改或扩充。举例来说，用来列出某些系统组件的条目个数（例如“硬件摘要”节中的Peripherals清单）可能不足以描述系统。你可以根据实际的需要多增加几页以便详细描述系统的特性。

欲取得空白的工作单副本，可到本书位于<http://www.embeddedtux.org/>的网站下载，有两种格式：PDF和OpenOffice。此外，还可以复印本附录提供的工作单。然而，请避免直接将数据填入本书，因为你可能会想要修改工作单，并在不同的项目里使用本书。

本附录其余的部分将会描述此工作单中每一节的细节。尽管大多数字段的用途很明显，不过某些字段可能需要稍加解释。你可以随时参考相关的章节。

项目标识符

这一节包含与嵌入式系统有关的高级信息。你可以在第一章找到该节需要的大部分信息。表A-1描述了“项目标识符”节的每个字段。

表 A-1：“项目标识符”各字段的说明

字段	说明
Name	此项目的名称
Internal ID	此项目在组织内部的识别代号（数字或字符串）
Project leader	此项目的主要负责人
Start date	此项目的发起时间
Expected completion date	此项目预期的完成时间
Project description	以高层的观点描述此项目
Type of system	第一章所提到的系统类型
Size（规模）	择一设定：小型（small）、中型（medium）或大型（large）。第一章对这三种规模的嵌入式Linux系统有完整的描述
Time constraints（时限）	择一设定：宽松（mild）或严格（stringent）。第一章讨论了这两种时限
Degree of user interaction（与用户交互的程度）	对系统用户接口复杂程度的测量。参见第一章所举的例子

表 A-1: “项目标识符” 各字段的说明 (续)

字段	说明
Are networking services offered or used?(网络能力)	择一设定: 有 (yes) 或无 (no)

硬件摘要

这一节包含“从软件观点来看硬件”的详细信息。该节需要的信息多半可以在嵌入式系统的规格书 (来自硬件部门或来自电路板和处理器厂商) 中找到。这些硬件信息对嵌入式 Linux 系统建立时的许多层面都很重要。表 A-2 描述了“硬件摘要”节的每个字段。正如前面所说, 可以根据实际的需要扩充 Peripherals 字段所包含的项目数, 这样你就可以填上系统所包含的每个外围设备。

表 A-2: “硬件摘要” 各字段的说明

字段	说明
Processor family	第三章所提到的处理器系列
Processor model	处理器系列里的处理器型号。举例来说, 如果该处理器属于 PowerPC 系列, 它的型号可能是 450、750、860 等等
Board type	制造商通常会对具有类似特性的电路板做系列或类型之分。如果电路板无此性质, 可以将此字段留白
Board model	电路板的型号或序号
RAM size	系统 RAM 的大小
RAM start and end addr	RAM 在物理地址空间中的位置
ROM/Flash size	系统 ROM 或 flash 的大小
ROM/Flash start and end addr	ROM/Flash 在物理地址空间中的位置
ROM/Flash model	ROM 或 flash 芯片的型号
Processor startup address	处理器取得第一个指令的地址
Disk storage type	如果使用的是 IDE 或 SCSI 设备, 或是可以充当磁盘的设备 (例如 CompactFlash 设备), 请填写此字段
Disk storage size	磁盘设备上的可用存储空间
Peripherals: Type	外围设备的类型: Ethernet 控制器、视频控制器、CAN 接口等等
Peripherals: Model	外围设备的型号或序号

表 A-2: “硬件摘要” 各字段的说明 (续)

字段	说明
Peripherals: Description	对外围设备特性的描述
Peripherals: Mem location	供外围设备使用的物理地址空间范围
Peripherals: ID	有些外围设备会有识别代号。举例来说, 假设此外围设备是一个 Ethernet 设备: 如果只生产一个这样的设备, 请写下该设备的 MAC 地址; 如果想生产许多这样的设备, 请写下这次生产所用到的 MAC 地址范围

开发工具

这一节的用途在于描述用来建立嵌入式系统的开发工具以及如何建立这些工具。该节需要的大部分信息与第四章提到的开发工具设置有关。你应该在建立工具的当时记录此处的信息, 这样才不会忘记。表 A-3 描述了“开发工具”节的每个字段。其中有许多字段对所有开发工具来说都是一样的。

表 A-3: “开发工具” 各字段的说明

字段	说明
Host type	“开发主机” 的类型 (参见第二章)
Tool: version	从项目网站下载的工具的正式版本
Tool: Special build flags	第四章并未提到用来建立工具的任何建立标志
Tool: Special configuration flags	第四章并未提到用来设定工具的任何配置标志
Tool: Configuration summary	摘要记录让工具得以建立完成的配置设定方式。只适用于 uClibc (参见第四章)
Tool: Patches/Changes	描述对工具进行的修补或修改
Editor/IDE	此项目的开发团队的成员所使用的编辑器或 IDE
Terminal emulator	此项目的开发团队的成员所使用的终端仿真程序
Notes	任何关于开发工具的设置和使用的附注

内核

这一节的用途在于描述嵌入式系统中使用内核的完整细节。该节的信息可以跟第五章提供的说明并用。表 A-4 描述了“内核”节的每个字段。

表 A-4: “内核” 各字段的说明

字段	说明
Version	到架构的主要下载网站取得的正式内核版本
Download location	你用来取得内核的 URL
Patches: Description	描述所使用的补丁
Patches: Download location	你用来取得补丁的 URL
Configuration file location	用来建立内核的配置文件所摆放位置的完整路径
Configuration summary	内核启用的重要配置选项的详细清单
Kernel failure handler description	描述你为系统所实现的内核panic处理例程。详情参见第五章“实地测试”一节

根文件系统

这一节的用途在于描述目标板根文件系统的完整细节。此处的信息可以跟第六章提供的说明并用。表 A-5 描述了“根文件系统”节的每个字段。此节的“*/dev device entries*”部分用来列出你建立的 */dev* 条目（涵盖第六章所提到的条目）。这一节的“*System applications*”部分用来列出你提供基本 Unix 服务时所使用的系统应用程序，例如 *BusyBox*、*TinyLogin* 和 *Embutils*。这一节的“*System initialization*”部分用来列出 *init* 启动的服务以及这些服务启动的方式。

表 A-5: “根文件系统” 各字段的说明

字段	说明
C library	嵌入式系统所使用的 C 链接库: <i>glibc</i> 、 <i>uClibc</i> 或 <i>diet libc</i>
C library components	被复制到目标板的 <i>/lib</i> 目录的 C 链接库组件（参见第六章）
<i>/dev device entries</i> : Name	此（文件系统）条目的名称
<i>/dev device entries</i> : Major nbr	此条目的设备主编号
<i>/dev device entries</i> : Minor nbr	此条目的设备次编号
<i>/dev device entries</i> : Used by	使用此条目的应用程序
System applications: Package	系统应用程序套件的名称
System applications: Version	套件的版本
System applications: Build config	摘要记录套件的配置。以 <i>BusyBox</i> 为例，应该列出你对预设配置做了哪些修改

表 A-5: “根文件系统” 各字段的说明（续）

字段	说明
System applications: Config file location	用来建立此套件的配置文件所摆放位置的完整路径
System initialization: Service	欲启动的服务或二进制文件
System initialization: Type of activation	<i>init</i> 的启动方式，例如 <i>askfirst</i> 、 <i>wait</i> 和 <i>once</i> ，完整的清单参见第六章

存储设备结构

这一节用来描述系统的存储设备的内容。主要存储设备一般指的是 ROM 或 flash 存储芯片，次要存储设备一般指的是磁盘或是附加的固态存储设备。这一节的内容应该与第七章和第八章的说明并用。表 A-6 描述了“存储设备结构”节的每个字段。

表 A-6: “存储设备结构” 各字段的说明

字段	说明
Development setup	正如第二章所说，这种设置用来将软件组件从主机转移到目标板
Storage device ^a content: Component	欲存储组件的类型。它可是一个引导加载程序、内核、引导参数或任何文件系统（包括根文件系统）
Storage device content: Size	为这个组件配置的存储空间
Storage device content: Location	组件在存储设备里的位置。对固态存储设备（例如 ROM 或 flash）而言，这就是设备的起始和结束地址。对磁盘存储设备而言，这就是开始和结束扇区
Storage device content: Dependency	会加载或使用此组件的其他组件。举例来说，根文件系统依赖内核、内核则依赖引导加载程序
Storage device content: Format	组件会以何种格式存储。对文件系统而言，这就是文件系统类型，例如 <i>ext2</i> 或 <i>JFFS2</i>

a. 尽管此处以通称（storage device）列出每个条目，不过正如前面所说，存储设备有许多类型。

引导加载程序的配置与使用

这一节包含引导加载程序的配置与使用的详细信息。此处的信息可以跟第九章的说明并用。表 A-7 描述了“引导加载程序的配置与使用”节的每个字段。

表 A-7: “引导加载程序的配置与使用” 各字段的说明

字段	说明
Package	此系统使用的引导加载程序
Version	套件的版本
Build configuration	描述建立配置
Setup procedure	将引导加载程序安装到目标板存储设备的方法。其中可以包括软件和硬件的操作
Boot options: Option	被设定进引导加载程序的引导选项
Boot options: Description	描述使用这个引导选项的结果
Default boot option	如果没有使用任何其他的选项，就启用默认的引导选项
Security	对引导加载程序进行锁定和解锁的安全程序以避免遭到用户修改

网络服务

这一节包含嵌入式系统提供的网络服务的相关细节。此处的信息可以跟第十章的说明并用。表 A-8 描述了“网络服务” 小节中的每个字段。

表 A-8: “网络服务” 各字段的说明

字段	说明
Main network service	此系统提供的主要网络服务
Service	所要提供的网络服务
Package	可提供此网络服务的套件
Version	套件的版本
Configuration summary	套件的配置摘要

定制项目软件

这一节包含了与定制项目软件有关的信息。欲取得此处的信息只需分析你自己的软件。此处不必记录软件的完整说明, 因为开发团队有自己的内部文件描述项目的结构与相关细节。此处只需概述定制软件与系统其他部分的关系。表 A-9 描述了“定制项目软件” 节的每个字段。

表 A-9: “定制项目软件” 各字段的说明

字段	说明
Main source repository	此项目源码的主要存储地点
Code maintainer	此项目源码的主要负责人
Location in target root filesystem	目标板的根文件系统上, 项目组件最终摆放位置的完整路径
Binary size	此项目中所有二进制文件会占用多少存储空间
Data size	此项目中所有数据最多会占用多少存储空间
Dependencies	此项目会与哪些其他的软件套件有依存关系。C 链接库有可能会列在此处
Initialization procedure	软件的启动方式。如果软件由 <code>init</code> 启动, 请将 <code>init</code> 列在此处

调试记录

这一节包含关于项目调试的信息。此处的信息可以跟第十一章的说明并用。调试是一个创造性的过程, 每个开发者完成的方式都不一样。因此, 这一节只作为摘要之用。团队极有可能需要在项目开发期间自己维护一份最新的缺陷清单。表 A-10 描述了“调试记录”节的每个字段。

表 A-10: “调试记录” 各字段的说明

字段	说明
Tools used	系统调试所使用工具的详细清单
Summary of major bugs found	摘要记录系统上所找到重要缺陷
System's fragilities	每个系统都有弱点。将这些弱点列在此处将可以让你以及参与系统开发的其他人注意到这些弱点可能引发的问题

附注

这一节用来加入与嵌入式系统有关的任何额外的细节; 只要你觉得参与项目或需要了解项目组件的组成对任何人有帮助就可以加入这些额外的细节。依实际的系统而定, 可能还会想要加入额外的节, 以便为目前这个工作单版本未涵盖在内的其他嵌入式系统记录相关细节。

Embedded Linux Systems WorkSheet

Project identification	
Name:	
Internal ID:	
Project leader:	
Start date:	
Expected completion date:	
Project description:	
Type of system:	Size:
Time constraints:	Degree of user interaction:
Are networking services offered or used?:	

Hardware summary				
Processor family:			Main contact:	
Processor model:				
Board type:				
Board model:				
RAM size:		Start addr:		End addr:
ROM/Flash size:		Start addr:		End addr:
ROM/Flash model:			Processor startup address:	
Disk storage type:		Disk storage size:		
Peripherals				
Type	Model	Description	Mem location	ID
			Start:	
			End:	
			Start:	
			End:	
			Start:	
			End:	

Development tools	
Host type:	Main contact:
binutils version: Special build flags: Patches / Changes: Patch location / Change description:	
gcc version: Special configuration flags: Patches / Changes: Patch location / Change description:	
glibc version: Special configuration flags: Patches / Changes: Patch location / Change description:	
uClibc version: Configuration summary: Patches / Changes: Patch location / Change description:	
diet libc version: Special build flags: Patches / Changes: Patch location / Change description:	
Editor / IDE:	
Terminal emulator:	
Notes:	

Kernel	
Version:	Main contact:
Download location:	
Patches	
Description	Download location
Configuration file location:	
Configuration summary:	
Kernel failure handler description:	

[illegible]

[illegible]

Custom project software
Main source repository:
Code maintainer:
Location in target root filesystem:
Binary size:
Data size:
Dependencies:
Initialization procedure:

Debug notes
Tools used:
Summary of major bugs found:
System's fragilities:

Additional notes

本书会适时参考外部数据。以下列出内容涵盖本书中心议题或周边议题的各种参考资料。

在线资源

在“嵌入式 Linux”越来越普及的同时，出现了许多志在帮助潜在使用者和采用者的网站。以下列出这类网站（采用英文字母顺序）：

All Linux Devices (<http://alllinuxdevices.com/>)

此网站提供有嵌入式 Linux 相关报导和消息的链接。被维护成 Internet.com 的 Linux 资源的一部分。

Embedded-Linux.de (<http://embedded-linux.de/>)

你可以在这个德语网站看到若干用于嵌入式 Linux 系统的主要源码套件（例如 BusyBox 和 uClibc）的更新消息。

LinuxAutomation (<http://www.linux-automation.de/>)

你可以在此网站看到在自动化应用中使用 Linux 的各种资源的链接。尽管此网站的首页使用的是德文，不过它提供有英文版的链接。此网站由 Robert Schwebel 负责维护。

LinuxDevices.com (<http://www.linuxdevices.com/>)

你可以在此网站看到许多业界的消息。你还可以看到许多关于开放源码和自由软件社群开发的文章，但颇具商业气息。此网站提供有许多介绍性的文档而且经常更新。它或许是最具吸引力的嵌入式 Linux 网站。

Linux Documentation Project (<http://www.tldp.org/>)

此网站是开放源码和自由软件套件相关的 HOWTO、FAQ 及其他文件的主要集散

库。就它的文件所涵盖议题的深度和广度来看,它或许是最重要的Linux资源之一。此网站由社群自己维护。

SiliconPenguin.com (<http://www.siliconpenguin.com/>)

此网站收集了嵌入式Linux相关资料的链接。

uCdot (<http://www.ucdot.org/>)

uClinux 使用者的新闻和社群网站。

以上所列的网站仅包含提供嵌入式Linux相关信息者,不过你可能会发现许多提供一般Linux信息的其他网站对你也很有用,可以在《Running Linux》中看到这类网站。

书籍

探讨Linux以及一般嵌入式系统的书籍相当多,以下仅列出一些可能对你有用的书目:

《Advanced Programming in the UNIX Environment》 作者Richard Stevens (出版商Addison-Wesley)

被许多人视为最重要的Unix程序设计书。如果有需要了解如何以Unix的心态来思考以及设计程序,这便是一本你需要阅读的书。Stevens的著作一般来说都是高度建议阅读的书。

《The Art of Designing Embedded Systems》 作者Jack Ganssle (出版商Newnes Press)

本书的风格与大多数其他的技术书籍不同,它融合了技术性的说明以及关于实际问题的经验建议。它捕捉到了大多数嵌入式系统设计者日常工作经验的精髓。Jack Ganssle在《Embedded Systems Programming》杂志有开辟一个经常性的专栏,他也是一位常常受到嵌入式系统研讨会邀请的发言者。

《Embedded Systems Design》 作者Arnold S. Berger (出版商CMP Books)

这是一本从硬件和软件的观点来介绍嵌入式系统设计的导读性书籍。如果对嵌入式系统的开发过程并不熟悉,将会发现本书对你很有帮助。

《Linux Device Drivers》 作者Alessandro Rubini 与Jonathan Corbet (出版商O'Reilly)

这是一本教你如何开发Linux设备驱动程序的典型教科书。本书由开放源码和自由软件社群受人尊敬的两位成员撰写。这是一本任何Linux设备驱动程序开发者必读的书。

《Running Linux》 作者Matt Welsh、Lar Kaufman、Terry Dawson 与Matthias Kalle Dalheimer (出版商O'Reilly)

阅读本书的读者不需要具备Linux或Unix的背景知识就能全完全控制安装和使用Linux的所有信息。我自己手边有本书的第一版,每当我忘了要如何完成Linux中

某件事时，我都会回头参考这本书。这是一本非常好的书，它提供了让你得以在嵌入式系统中善用 Linux 的大量背景信息。

《*Programming Embedded Systems in C and C++*》作者 Michael Barr (出版商 O'Reilly)

这本导读性的书籍涵盖嵌入式软件开发的基础知识，并且对用来开发嵌入式系统的许多软件技巧提供了深入剖析。

《*Understanding the Linux Kernel*》作者 Daniel Bovet 与 Marco Cesati (出版商 O'Reilly)

尽管这些年来市面上出版了若干探讨 Linux 内核内部的书籍，不过本书无论在研究的深度上或是在结构的安排上均较为突出。本书目前涵盖 Linux 2.4 稳定版。

刊物

尽管写作本书时尚未出现任何以嵌入式 Linux 为主的刊物，不过有许多探讨其他主题的刊物有时会涵盖 Linux 在嵌入式系统中的使用：

《*Embedded Systems Programming*》(<http://www.embedded.com/mag.html>)

这是一本专为嵌入式软件设计人员发行的杂志。它对各种特殊的议题提供了许多令人感兴趣以及深入的文章。经过审核的读者可以免费订阅此杂志。强烈建议读者花点时间订阅此刊物。

《*Linux Journal*》(<http://www.linuxjournal.com/>)

这是最老牌以及信誉最卓著的 Linux 刊物。《Linux Journal》的出版商还发行了《Embedded Linux Journal》专门探讨在嵌入式系统中使用 Linux 的课题，但后来停刊了，改为在每期《Linux Journal》开辟经常性的 Embedded 专栏。

《*Linux Magazine*》(<http://www.linuxmagazine.com/>)

这是另一份信誉最卓著的 Linux 刊物。内容涵盖 Linux 的各种应用。

《*Linux Magazine France*》(<http://www.linuxmag-france.org/>)

这是一份法语刊物，所提供的全都是关于各种开放源码和自由软件套件的文章。这些文章在论述如何设定各种命令和服务时通常会提供许多的程序范例以及提示。

组织

正如第一章所说，有若干组织的活动涉及 Linux 在嵌入式系统中的使用：

- Embedded Linux Consortium (<http://www.embedded-linux.org/>)
- Emblix (<http://www.emblix.org/>)

- Filesystem Hierarchy Standard Group (<http://www.pathname.com/fhs/>)
- Free Software Foundation (<http://www.fsf.org/>)
- Free Standards Group (<http://www.freestandards.org/>)
- Linux Standard Base (<http://www.linuxbase.org/>)
- OpenGroup (<http://www.opengroup.org/>)
- Real-Time Linux Foundation (<http://www.realltimelinuxfoundation.org/>)
- TV Linux Alliance (<http://www.tvlinuxalliance.org/>)

Linux 以及面向开放源码的硬件计划

FreeIO 计划 (<http://www.freeio.org/>)

FreeIO (Free Hardware Resources for the Free Software Community) 致力于硬件原理图和设计图的开发与发行 (采用 GNU GPL 的许可条款)。此网站目前已经提供有若干硬件设计图以及相关的 Linux 驱动程序。

LART 计划 (<http://www.lart.tudelft.nl/>)

此计划的目标在开发以 StrongARM 为基础的嵌入式电路板供 Linux 运行。此网站目前提供有若干电路板示意图、扩展模块以及软件。

MyLinux 计划 (<http://www.azpower.com/mylinux/>)

此计划的目标在开发以 SuperH 为基础的类似 PDA 的嵌入式系统供 Linux 运行。此网站提供了该计划的细节以及照片。

Opencores.ORG 计划 (<http://www.opencores.org/>)

此计划收集了开发 Intellectual Property (知识产权, IP) core (内核) 各种计划, 它的发行采用的是 GNU GPL 许可条款。此网站目前已经提供了许多基本组件。

Simputer 计划 (<http://www.simputer.org/>)

致力开发廉价的参考硬件平台供 Linux 运行。

TuxScreen 计划 (<http://www.tuxscreen.net/>)

原为 Philips 的一项产品, TuxScreen 是个以 StrongARM 为基础的平台, 它包括了一个话机、一个屏幕以及一个完整的键盘。尽管此产品已无现货可供应, 不过此网站所提供的示意图对其他计划可能会很有用。

uClinux boards 计划 (<http://www.uclinux.org/>)

这是一个“最初就将目标明确地定位在构建可供 Linux 运行的嵌入式系统”的硬件计划。Linux 的 MMU-less 移植便源于该计划。

重要的版权声明

开源和自由软件的使用和流通受限于若干广为人知的授权书（参见第一章的说明）。尽管如此，与 Linux 授权相关的话题仍不断浮现造成混淆。这些不确定性起因于 Linux 核心的发行采用的是 GNU GPL 的授权条款。

其中，Linus Torvalds 和其他核心开发者致力阐明核心授权的限制和范围。该附录将会提供 Linus 和其他核心开发者对核心授权的三个层面（非 GPL 应用程序的使用、纯二进制模块的使用、与核心原码有关的一般授权问题）所发表的看法。

将 User-Space 应用程序排除在核心的 GPL 授权之外

为避免对 Linux 核心上所执行的应用程序的授权状态造成混淆，Linus Torvalds 在核心的授权书中加入了如下的前言：

```
NOTE! This copyright does *not* cover user programs that use kernel
services by normal system calls - this is merely considered normal use
of the kernel, and does *not* fall under the heading of "derived work".
Also note that the GPL below is copyrighted by the Free Software
Foundation, but the instance of code that it refers to (the Linux
kernel) is copyrighted by me and others who actually wrote it.
```

```
Also note that the only valid version of the GPL as far as the kernel
is concerned is _this_ license (ie v2), unless explicitly otherwise
stated.
```

Linus Torvalds

对二进制核心模块所做的声明

不断有人质疑不以GPL授权条款发行的可载入核心模块。有许多公司已经在提供这种二进制模块，而且业界有许多人坚持主张这种模块的合法性。然而，有许多Linux核心的开发者挺身而出强烈反对这件事。Linus Torvalds和Alan Cox曾在Linux kernel邮递论坛上对二进制模块的使用发表过以下的看法：

Linus在Kernel Interface Thread所发表第一封信

```
From:      torvalds@transmeta.com (Linus Torvalds)
Subject:   Re: Kernel interface changes (was Re: cdrecord problems on
Date:      1999-02-05 7:13:23
```

```
In article <36bab0c7.394438@mail.cloud9.net>,
John Alvord <jalvo@cloud9.net> wrote:
>On Thu, 4 Feb 1999 22:37:06 -0500 (EST), "Theodore Y. Ts'o"
><tytso@MIT.EDU> wrote:
>>
>>And as a result, I've seen more than a few MIT users decide to give up
>>on Linux and move over to NetBSD. I think this is bad, and I'm hoping
>>we can take just a little bit more care in the 2.2 series than we did in
>>the 2.0 series. Is that really too much to ask?
```

Yes. I think it is. I will strive for binary compatibility for modules, but I expect that it will be broken. It's just too easy to have to make changes that break binary-only modules, and I have too little incentive to try to avoid it.

If people feel this is a problem, I see a few alternatives:

- don't use stuff with binary-only modules. Just say no.
- work hard at making a source-version of the thing available (it doesn't have to be under the GPL if it's a module, but it has to be available as source so that it can be recompiled).
- don't upgrade
- drop Linux

```
>I suggest we treat binary compatibility problems as bugs which need to
>be resolved during the 2.2 lifetime. Even with all care, some changes
>will occur because of mistakes... if we cure them, there will be
>limited impact to users.
```

It's often not mistakes. Things sometimes have to change, and I personally do not care for binary-only modules enough to even care. If people want to use Linux, they have to live with this. In 2.2.x, the basics may be stable enough that maybe the binary module interface won't actually change. I don't know. That would be good, but if it is not to be, then it is not to be.

I allow binary-only modules. I allow them because I think that sometimes I cannot morally require people to make sources available to

projects like AFS where those sources existed before Linux. HOWEVER, that does not mean that I have to like AFS as a binary-only module.

Quite frankly, I hope AFS dies a slow and painful death with people migrating to better alternatives (coda, whatever). Or that somebody makes an AFS client available in source form, either as a clone or through the original people.

As it is, what has AFS done for me lately? Nothing. So why should I care?

Linus

Linus 在 Kernel Interface Thread 所发表的第 2 封信

From: torvalds@transmeta.com (Linus Torvalds)?
Subject: Re: Kernel interface changes (was Re: cdrecord problems on
Date: 1999-02-07 8:15:24

In article <79g5bu\$spd\$1@palladium.transmeta.com>,

H. Peter Anvin <hpa@transmeta.com> wrote:

>

>* Linus Torvalds has no interest whatsoever in developing such a
> plug-in ABI. Someone else is welcome to do it.

No, it's even more than that.

I refuse to even consider tying my hands over some binary-only module.

Hannu Savolainen tried to add some layering to make the sound modules more "portable" among Linux kernel versions, and I disliked it for two reasons:

- extra layers decrease readability, and sometimes make for performance problems. The readability thing is actually the larger beef I had with this: I just don't want to see drivers start using some strange wrapper format that has absolutely nothing to do with how they work.
- I want people to expect that interfaces change. I want people to know that binary-only modules cannot be used from release to release. I want people to be really really REALLY aware of the fact that when they use a binary-only module, they tie their hands.

Note that the second point is mainly psychological, but it's by far the most important one.

Basically, I want people to know that when they use binary-only modules, it's **THEIR** problem. I want people to know that in their bones, and I want it shouted out from the rooftops. I want people to wake up in a cold sweat every once in a while if they use binary-only modules.

Why? Because I'm a prick, and I want people to suffer? No.

Because I know that I will eventually make changes that break modules.

And I want people to expect them, and I never EVER want to see an email in my mailbox that says "Damn you, Linus, I used this binary module for over two years, and it worked perfectly across 150 kernel releases, and Linux-5.6.71 broke it, and you had better fix your kernel".

See?

I refuse to be at the mercy of any binary-only module. And that's why I refuse to care about them - not because of any really technical reasons, not because I'm a callous bastard, but because I refuse to tie my hands behind my back and hear somebody say "Bend Over, Boy, Because You Have It Coming To You".

I allow binary-only modules, but I want people to know that they are only ever expected to work on the one version of the kernel that they were compiled for. Anything else is just a very nice unexpected bonus if it happens to work.

And THAT, my friend, is why when somebody complains about AFS, I tell them to go screw themselves, and not come complaining to me but complain to the AFS boys and girls. And why I'm not very interested in changing that.

Linus

Alan Cox 在 Kernel Hooks Thread 所发表的信件

这封是针对 *Theodore Ts'O* 的回信。

From: Alan Cox <alan@lxorguk.ukuu.org.uk>
Subject: Re: [ANNOUNCE] Generalised Kernel Hooks Interface (GKHI)
Date: 2000-11-09 14:26:33

> Actually, he's been quite specific. It's ok to have binary modules as
> long as they conform to the interface defined in /proc/ksyms.

What is completely unclear is if he has the authority to say that given that there is code from other people including the FSF merged into the tree.

I've taken to telling folks who ask about binary modules to talk to their legal department. The whole question is simply too complicated for anyone else to work on.

Alan

Linus 在 Security Hooks License Thread 所发表的第 1 封信

From: Linus Torvalds <torvalds@transmeta.com>
Subject: Re: [PATCH] make LSM register functions GPLonly exports
Date: 2002-10-17 17:08:19

Note that if this fight ends up being a major issue, I'm just going to

remove LSM and let the security vendors do their own thing. So far

- I have not seen a lot of actual usage of the hooks
- seen a number of people who still worry that the hooks degrade performance in critical areas
- the worry that people use it for non-GPL'd modules is apparently real, considering Crispin's reply.

I will re-iterate my stance on the GPL and kernel modules:

There is NOTHING in the kernel license that allows modules to be non-GPL'd.

The only thing that allows for non-GPL modules is copyright law, and in particular the "derived work" issue. A vendor who distributes non-GPL modules is not protected by the module interface per se, and should feel very confident that they can show in a court of law that the code is not derived.

The module interface has NEVER been documented or meant to be a GPL barrier. The COPYING clearly states that the system call layer is such a barrier, so if you do your work in user land you're not in any way beholden to the GPL. The module interfaces are not system calls: there are system calls used to install them, but the actual interfaces are not.

The original binary-only modules were for things that were pre-existing works of code, ie drivers and filesystems ported from other operating systems, which thus could clearly be argued to not be derived works, and the original limited export table also acted somewhat as a barrier to show a level of distance.

In short, Crispin: I'm going to apply the patch, and if you as a copyright holder of that file disagree, I will simply remove all of the LSM code from the kernel. I think it's very clear that a LSM module is a derived work, and thus copyright law and the GPL are not in any way unclear about it.

If people think they can avoid the GPL by using function pointers, they are WRONG. And they have always been wrong.

Linus

Linus 在 Security Hooks License Thread 所发表的第 2 封信

From: Linus Torvalds <torvalds@transmeta.com>
Subject: Re: [PATCH] make LSM register functions GPLonly exports
Date: 2002-10-17 17:25:12

On Thu, 17 Oct 2002, Linus Torvalds wrote:

>

> If people think they can avoid the GPL by using function pointers, they
> are WRONG. And they have always been wrong.

Side note: it should be noted that legally the GPLONLY note is nothing but

a strong hint and has nothing to do with the license (and only matters for the `_enforcement_` of said license). The fact is:

- the kernel copyright requires the GPL for derived works anyway.

if a company feels confident that they can prove in court that their module is not a derived work, the GPL doesn't matter `_anyway_`, since a copyright license at that point is meaningless and wouldn't cover the work regardless of whether we say it is GPLONLY or not.

(In other words: for provably non-derived works, whatever kernel license we choose is totally irrelevant)

So the GPLONLY is really a big red warning flag: "Danger, Will Robinson".

It doesn't have any real legal effect on the meaning of the license itself, except in the sense that it's another way to inform users about the copyright license (think of it as a "click through" issue - GPLONLY forces you to "click through" the fact that the kernel is under the GPL and thus derived works have to be too).

Clearly "click through" `_has_` been considered a legally meaningful thing, in that it voids the argument that somebody wasn't aware of the license. It doesn't change what you can or cannot do, but it has some meaning for whether it could be wilful infringement or just honest mistake.

Linus

Linus Torvalds 对核心的授权范围提出澄清

Linus Torvalds对核心的授权以及这项授权如何应用在外程序码提供了非常详细的说明:

Feel free to post/add this. I wrote it some time ago for a corporate lawyer who wondered what the "GPL exception" was. Names and companies removed not because I think they are ashamed, but because I don't want people to read too much into them.

Linus

. --

Date: Fri, 19 Oct 2001 13:16:45 -0700 (PDT)
 From: Linus Torvalds <torvalds@transmeta.com>
 To: Xxxx Xxxxxx <xxxxx@xxx.xxx.com>
 Subject: Re: GPL, Richard Stallman, and the Linux kernel

[This is not, of course, a legal document, but if you want to forward it to anybody else, feel free to do so. And if you want to argue legal points with me or point something out, I'm always interested. To a point ;-]

On Fri, 19 Oct 2001, Xxxx Xxxxxx wrote:

>
> I've been exchanging e-mail with Richard Stallman for a couple of
> weeks about the finer points of the GPL.

I feel your pain.

> I've have spent time pouring through mailing list archives, usenet,
> and web search engines to find out what's already been covered about
> your statement of allowing dynamically loaded kernel modules with
> proprietary code to co-exist with the Linux kernel. So far I've
> been unable to find anything beyond vague statements attributed to
> you. If these issues are addressed somewhere already, please refer
> me.

Well, it really boils down to the equivalent of "_all_ derived modules
have to be GPL'd". An external module doesn't really change the GPL in
that respect.

There are (mainly historical) examples of UNIX device drivers and some
UNIX filesystems that were pre-existing pieces of work, and which had
fairly well-defined and clear interfaces and that I personally could not
really consider any kind of "derived work" at all, and that were thus
acceptable. The clearest example of this is probably the AFS (the Andrew
Filesystem), but there have been various device drivers ported from SCO
too.

> Issue #1
> = = = = =
> Currently the GPL version 2 license is the only license covering the
> Linux kernel. I cannot find any alternative license explaining the
> loadable kernel module exception which makes your position difficult
> to legally analyze.
>
> There is a note at the top of www.kernel.org/pub/linux/kernel/COPYING,
> but that states "user programs" which would clearly not apply to
> kernel modules.
>
> Could you clarify in writing what the exception precisely states?

Well, there really is no exception. However, copyright law obviously
hinges on the definition of "derived work", and as such anything can
always be argued on that point.

I personally consider anything a "derived work" that needs special hooks
in the kernel to function with Linux (ie it is _not_ acceptable to make a
small piece of GPL-code as a hook for the larger piece), as that obviously
implies that the bigger module needs "help" from the main kernel.

Similarly, I consider anything that has intimate knowledge about kernel
internals to be a derived work.

What is left in the gray area tends to be clearly separate modules: code
that had a life outside Linux from the beginning, and that do something
self-contained that doesn't really have any impact on the rest of the

kernel. A device driver that was originally written for something else, and that doesn't need any but the standard UNIX read/write kind of interfaces, for example.

> Issue #2

> = = = = =

> I've found statements attributed to you that you think only 10% of
> the code in the current kernel was written by you. By not being the
> sole copyright holder of the Linux kernel, a stated exception to
> the GPL seems invalid unless all kernel copyright holders agreed on
> this exception. How does the exception cover GPL'd kernel code not
> written by you? Has everyone contributing to the kernel forfeited
> their copyright to you or agreed with the exception?

Well, see above about the lack of exception, and about the fundamental gray area in any copyright issue. The "derived work" issue is obviously a gray area, and I know lawyers don't like them. Crazy people (even judges) have, as we know, claimed that even obvious spoofs of a work that contain nothing of the original work itself, can be ruled to be "derived".

I don't hold views that extreme, but at the same time I do consider a module written for Linux and using kernel infrastructures to get its work done, even if not actually copying any existing Linux code, to be a derived work by default. You'd have to have a strong case to not consider your code a derived work..

> Issue #3

> = = = = =

> This issue is related to issue #1. Exactly what is covered by the
> exception? For example, all code shipped with the Linux kernel
> archive and typically installed under /usr/src/linux, all code under
> /usr/src/linux except /usr/src/linux/drivers, or just the code in
> the /usr/src/linux/kernel directory?

See above, and I think you'll see my point.

The "user program" exception is not an exception at all, for example, it's just a more clearly stated limitation on the "derived work" issue. If you use standard UNIX system calls (with accepted Linux extensions), your program obviously doesn't "derive" from the kernel itself.

Whenever you link into the kernel, either directly or through a module, the case is just a lot more muddy. But as stated, by default it's obviously derived - the very fact that you need to do something as fundamental as linking against the kernel very much argues that your module is not a stand-alone thing, regardless of where the module source code itself has come from.

> Issue #4

> = = = = =

> This last issue is not so much a issue for the Linux kernel
> exception, but a request for comment.
>

> Richard and I both agree that a "plug-in" and a "dynamically
> loaded kernel module" are effectively the same under the GPL.

Agreed.

The Linux kernel modules had (a long time ago), a more limited interface, and not very many functions were actually exported. So five or six years ago, we could believably claim that "if you only use these N interfaces that are exported from the standard kernel, you've kind of implicitly proven that you do not need the kernel infrastructure".

That was never really documented either (more of a guideline for me and others when we looked at the "derived work" issue), and as modules were more-and-more used not for external stuff, but just for dynamic loading of standard linux modules that were distributed as part of the kernel anyway, the "limited interfaces" argument is no longer a very good guideline for "derived work".

So these days, we export many internal interfaces, not because we don't think that they would "taint" the linker, but simply because it's useful to do dynamic run-time loading of modules even with standard kernel modules that are supposed to know a lot about kernel internals, and are obviously "derived works"...

> However we disagree that a plug-in for a GPL'd program falls
> under the GPL as asserted in the GPL FAQ found in the answer:
> <http://www.gnu.org/licenses/gpl-faq.html#GPLAndPlugins>.

I think you really just disagree on what is derived, and what is not. Richard is very extreme: anything that links is derived, regardless of what the arguments against it are. I'm less extreme, and I bet you're even less so (at least you might like to argue so).

> My assertion is that plug-ins are written to an interface, not a
> program. Since interfaces are not GPL'd, a plug-in cannot be GPL'd
> until the plug-in and program are placed together and run. That is
> done by the end user, not the plug-in creator.

I agree, but also disrespectfully disagree ;)

It's an issue of what a "plug-in" is - is it a way for the program to internally load more modules as it needs them, or is it meant to be a public, published interface.

For example, the "system call" interface could be considered a "plug-in interface", and running a user mode program under Linux could easily be construed as running a "plug-in" for the Linux kernel. No?

And there, I obviously absolutely agree with you 100%: the interface is published, and it's meant for external and independent users. It's an interface that we go to great lengths to preserve as well as we can, and it's an interface that is designed to be independent of kernel versions.

But maybe somebody wrote his program with the intention to dynamically load "actors" as they were needed, as a way to maintain a good modularity, and to try to keep the problem spaces well-defined. In that case, the

"plug-in" may technically follow all the same rules as the system call interface, even though the author doesn't intend it that way.

So I think it's to a large degree a matter of intent, but it could arguably also be considered a matter of stability and documentation (ie "require recompilation of the plug-in between version changes" would tend to imply that it's an internal interface, while "documented binary compatibility across many releases" implies a more stable external interface, and less of a derived work).

Does that make sense to you?

> I asked Richard to comment on several scenarios involving plug-ins
> explain whether or not they were in violation of the GPL. So far he
> as only addressed one and has effectively admitted a hole. This is
> the one I asked that he's responded to:

> [A] non GPL'd plug-in writer writes a plug-in for a non-GPL'd
> program. Another author writes a GPL'd program making the
> first author's plug-ins compatible with his program. Are now
> the plug-in author's plug-ins now retroactively required to be
> GPL'd?

>

> His response:

> No, because the plug-in was not written to extend this program.

>

> I find it suspicious that whether or not the GPL would apply to the
> plug-in depends on the mindset of the author.

The above makes no sense if you think of it as a "plug in" issue, but it makes sense if you think of it as a "derived work" issue, along with taking "intent" into account.

I know lawyers tend to not like the notion of "intent", because it brings in another whole range of gray areas, but it's obviously a legal reality.

Ok, enough blathering from me. I'd just like to finish off with a few comments, just to clarify my personal stand:

- I'm obviously not the only copyright holder of Linux, and I did so on purpose for several reasons. One reason is just because I hate the paperwork and other cr*p that goes along with copyright assignments.

Another is that I don't much like copyright assignments at all: the author is the author, and he may be bound by my requirement for GPL, but that doesn't mean that he should give his copyright to me.

A third reason, and the most relevant reason here, is that I want people to know that I cannot control the sources. I can write you a note to say that "for use XXX, I do not consider module YYY to be a derived work of my kernel", but that would not really matter that much. Any other Linux copyright holder might still sue you.

This third reason is what makes people who otherwise might not trust me realize that I cannot screw people over. I am bound by the same agreement that I require of everybody else, and the only special status

I really have is a totally non-legal issue: people trust me.

(Yes, I realize that I probably would end up having more legal status than most, even apart from the fact that I still am the largest single copyright holder, if only because of appearances)

- I don't really care about copyright law itself. What I care about is my own morals. Whether I'd ever sue somebody or not (and quite frankly, it's the last thing I ever want to do - if I never end up talking to lawyers in a professional context, I'll be perfectly happy. No disrespect intended) will be entirely up to whether I consider what people do to me "moral" or not. Which is why intent matters to me a lot - both the intent of the person/corporation doing the infringement, and the intent of me and others in issues like the module export interface.

Another way of putting this: I don't care about "legal loopholes" and word-wrangling.

- Finally: I don't trust the FSF. I like the GPL a lot - although not necessarily as a legal piece of paper, but more as an intent. Which explains why, if you've looked at the Linux COPYING file, you may have noticed the explicit comment about "only `_this_` particular version of the GPL covers the kernel by default".

That's because I agree with the GPL as-is, but I do not agree with the FSF on many other matters. I don't like software patents much, for example, but I do not want the code I write to be used as a weapon against companies that have them. The FSF has long been discussing and is drafting the "next generation" GPL, and they generally suggest that people using the GPL should say "v2 or at your choice any later version".

Linux doesn't do that. The Linux kernel is v2 ONLY, apart from a few files where the author put in the FSF extension (and see above about copyright assignments why I would never remove such an extension).

The "v2 only" issue might change some day, but only after all documented copyright holders agree on it, and only after we've seen what the FSF suggests. From what I've seen so far from the FSF drafts, we're not likely to change our v2-only stance, but there might of course be legal reasons why we'd have to do something like it (ie somebody challenging the GPLv2 in court, and part of it to be found unenforceable or similar would obviously mean that we'd have to reconsider the license).

Linus

PS. Historically, binary-only modules have not worked well under Linux, quite regardless of any copyright issues. The kernel just develops too quickly for binary modules to work well, and nobody really supports them. Companies like RedHat etc tend to refuse to have anything to do with binary modules, because if something goes wrong there is nothing they can do about it. So I just wanted to let you know that the `_legal_` issue is just the beginning. Even though you probably don't personally care ;)

源代码索引

前言

本索引让读者方便查找特定词汇在本书里的页码、快速找到所需的内容。传统上，中文书没有编制索引的惯例，因为中文不像英文那样有一套公认而且大家都知道的排序规则（字母顺序）。然而，这是一本工具书，为了方便读者查阅，索引是不可或缺的。为了兼顾“查阅”与“中文化”，我们决定沿袭英文版的编排格式，也就是依照英文字母顺序排列所有条目，所以，我们保留所有原文，如此读者才能快速找到想找的条目，而中文是以辅助说明的方式出现的。

然而，使用英文字母顺序编排意味着读者必须先知道英文词汇原文才能顺利找到其所在的页码。例如，假设读者想知道本书哪几页有提到“域名服务”，那么，你必须先知道其原文是“Domain Name Service”，或是知道其缩写是“DNS”，然后才能推断此条目应该是编在“D”小节。如果读者觉得这样不方便，我们为此感到抱歉，因为我们还没有找到一套大家都公认的中文排序规则，如果我们像编字典那样使用首字笔划顺序来排列，除了不方便查找之外，还必须面临一词多译的问题。例如，有人习惯将“serial port”翻译成“串行端口”，但也有人将它翻译成“顺序端口”，如果你不知道本书采用哪一种译词，要第一次就顺利找到“serial port”的页码，唯一的办法是碰运气。如果运气不好，你必须同时知道 serial port 的每一种可能译法才有机会找到。

格式说明

所有条目都是依照英文字母顺序排列。举一个实例说明如何使用本篇索引，以及索引条目的编排格式。如果你想知道本书哪儿页提到“performance tuning（性能调校）”，则必须先翻到“P”小节，然后你会看到：

performance tuning (性能调校)
 basic steps (基本步骤), 97
 capacity planning (容量规划), 107-111
 External (外部的), 102
 Internal (内部的), 103

如你所见, 我们以缩排格式来表示各项信息。通常, 从最左侧逐层往右读, 可以得到一个符合文法的完整句子或词汇; 不过, 并非每次都能这样, 有时候你得到的只是特定概念的关键词而已, 例如: “performance tuning、基本步骤”。所以, 这段索引是这样解读的: 第 97 页提到了“性能调校的基本步骤”, 在 103 页提到了“内部的性能调校”。

最后, 在不妨碍查阅的前提下, 对于第二层与第三层的条目, 我们会予以中文化。

希望本节的说明有助于读者使用本篇索引。如果读者对本公司书籍的索引方式有任何意见, 请用 email 告诉我们, 好让我们知道如何改进。

符号

\ (backward slash)(反斜线), 278
 : (colon)(冒号), 281
 - (hyphen)(连字符), 125
 | (pipe)(管道符号), 252
 ; (semicolon)(分号), 279
 / (slash)(斜线), 147, 197, 204
 ~> string (字符串), 285

A

accelerator control (加速器控制), 18, 19
 access rights (存取权限), 157
 ad hoc scripts (特制命令脚本), 249-252
 Ada Core Technologies Inc. (ACT), 155
 Ada for GNU/Linux Team (ALT), 155
 Ada programming language (Ada 程序语言), 154
 add_mtd_device() function (add_mtd_device() 函数), 92
 addr2line utility (addr2line 公用程序), 126
 Adeos nanokernel (Adeos 超微内核), 37
 Advanced RISC Machine (见 ARM)
 Aegis project (Aegis 计划), 150
 AF_BLUETOOTH socket type, 105
 Affix stack (Affix 堆栈), 105
 agents (代理程序), 297, 298

ahead-of-time (AOT) compiler (预先编译器), 150
 ALERT syslog level (ALERT 系统日志等级), 112
 ALICE(Automation Light Interface Control Environment) project (ALICE 计划), 87
 allinone binary (allinone 二进制)(embutils), 198
 ALSA(Advanced Linux Sound Architecture), 89
 Anjuta (IDE), 156
 Apache servers (Apache 服务器), 309, 312
 APIs (应用程序编程接口)
 DAQ hardware interfaces (DAQ 硬件接口), 84
 filesystem access (文件系统的存取), 58
 low-level services (低阶的服务), 58
 Open Sound System, 89
 parallel ports (并行端口), 83
 PCI bus (PCI 总线), 72
 portability (可移植性), 57
 Apple
 FireWire trademark (FireWire 商标), 79
 (另见 PowerPC)
 applications (应用程序)
 coverage analysis (覆盖范围分析), 331-333

- debugging with gdb (用 gdb 进行调试), 317-321
- dynamically linking libraries (动态链接库), 59
- GNU C library usage (GNU C 链接库的使用), 59
- GPL and (GPL 与), 32
- linking proprietary (链接所有权), 33
- linking to C library (链接 C 链接库), 133
- linking to uClibc library (链接 uClibc 链接库), 146
- linking with diet libc (链接 diet libc), 148
- root filesystem and (根文件系统与), 198
- root privileges and (root 权限与), 181
- worksheet (工作单), 351
(另见 system applications)
- ar utility (ar 公用程序), 126, 135
- ARCH variable (ARCH 变量) (make 命令), 125, 167, 171
- architecture embedded Linux system (架构, 嵌入式 Linux 系统) 30, 31
- ARCnet (Attached Resource Computer Network), 109, 110
- ARM (Advanced RISC Machine) processor (ARM 处理器)
 - architecture overview (架构概述) 59, 60
 - bootloader comparison (引导加载程序的比较), 256
 - diet libc support (对 diet libc 的支持), 146
 - embedded system survey (嵌入式系统的调查), 22
 - embutils, 197
 - GNU toolchain (GNU 工具链), 117
 - kernel considerations (内核的考虑), 164, 167, 169, 173
 - U-Boot and (U-Boot 与), 257
 - UI modules and (UI 模块与), 44
- ARMBoot project (ARMBoot 计划), 260
- as (GNU assembler) utility (as (GNU 汇编器) 公用程序), 126, 135
- ASCII
 - Modbus messaging format (Modbus 消息格式), 110
- ATA (ARCnet Trade Association), 109
- ATA (AT Attachment), 97
- ATA-ATAPI (IDE) hardware support (对 ATA-ATAPI 硬件的支持), 97-99
- ATAPI (ATA Packet Interface), 97
- Attached Resource Computer NETwork (ARCnet), 109, 110
- ATV (Automatic Transfer Vehicle)(自动运输工具), 20
- authentication (认证), 320
- authorization secure (授权、安全), 296
- B**
 - .bb files (.bb 文件), 332
 - .bbg files (.bbg 文件), 332
 - BBT (Bad Block Table)(坏区表), 219, 220, 223
 - BDM debugger (BDM 调试器), 48, 56, 343
 - BDM4GDB project (BDM4GDB 计划), 343
 - /bin directory (/bin 目录), 135, 180, 181, 198
 - binaries (二进制: 可执行文件)
 - downloading to flash (下载至闪存设备), 283-285
 - sections for debugging (用于调试的段), 317
 - strip command (strip 命令), 319
 - U-Boot, 280-281
 - BIN_GROUP variable (BIN_GROUP 变量) (make 命令), 200
 - BIN_OWNER variable (BIN_OWNER 变量) (make 命令), 200
 - binutils (binary utilities)(二进制公用程序)
 - gcc cautions (gcc 的注意事项), 136
 - GPL license (GPL 授权书), 32
 - resources (资源), 122
 - setting up (设置), 126-128
 - Unix systems and (Unix 系统与), 52
 - version considerations (版本的考虑), 116
 - BIOS
 - interrupt handlers and (中断处理例程与), 271
 - LILO and (LILO 与), 268
 - ROM chips and (ROM 芯片与), 258
 - system startup process (系统启动过程), 223
 - TSR program and (TSR 程序与), 270
 - Blackdown project (Blackdown 计划), 66, 149, 150
 - blob bootloader (blob 引导加载程序), 256, 259
 - block devices (块设备), 236, 242
 - BlueCat (LynuxWorks), 29
 - BlueDrekar stack (BlueDrekar 堆栈), 105
 - Bluetooth (蓝牙), 104
 - Ericsson blip and (Ericsson blip 与), 19
 - hardware support (所支持的硬件), 104-107

- kernel support options (内核支持选项), 167
- unlisted features (未列出的特性), 168
- BlueZ stack (BlueZ 堆栈), 105, 106
- Boa, 309-311
- book resources (书籍资源), 362, 363
- boot configuration (引导配置), 59-61
- /boot directory (/boot 目录), 179
- boot scripts (引导脚本), 279
- bootable DOS diskette (可引导的 DOS 软盘), 219, 220
- booting (引导)
 - basics of (的基础), 59
 - BOOTP/DHCP/TFTP/NFS, 282-283
 - CF devices (CF 设备), 287-290
 - diskless systems (无盘系统), 258
 - from DOC (自 DOC), 224
 - hard disks and (硬盘与), 231
 - network boot (网络引导), 257, 261-265
 - RAM disk (RAM 磁盘), 286-287
 - from ROM (自 ROM), 258
 - system reboot (系统重新引导), 176, 202
 - U-Boot and (U-Boot 与), 275-277
- bootldr bootloader (bootldr 引导加载程序), 256, 259
- bootloaders (引导加载程序)
 - ATA-IDE limitations (ATA-IDE 的限制), 98
 - boot configurations (引导配置), 59-61
 - comparison (比较), 255-257
 - DOS method installation (DOS 的安装方法), 220
 - example (范例), 63
 - GRUB and DOC devices (GRUB 和 DOC 设备), 270-274
 - importance of (的重要性), 254
 - installing (安装), 223, 224
 - LILO with disk and CF devices (使用磁盘和 CF 设备的 LILO), 266-269
 - minicom constraints (minicom 的限制), 159
 - mounting filesystem (挂载文件系统), 265-266
 - partitions and (分区与), 219
 - server setup for network boot (网络引导的服务器规划), 261-265
 - setting up (设置), 45, 47
 - SPL as (以 SPL 做为), 223
 - system startup component (系统启动组件), 59
 - worksheet (工作单), 350
 - :另见 specific bootloaders)
- bootm command (bootm 命令), 289
- BOOTP
 - booting with (以 (BOOTP) 引导), 282-283
 - GRUB network boot (GRUB 网络引导), 257
 - network boot (网络引导), 61, 261
 - SYSM module and (SYSM 模块与), 43
- bootp command (bootp 命令), 282
- BSD license (BSD 授权书)
 - inetd, 294
 - strace, 323
 - rhttpd, 309
 - xinetd, 296
- .bss section (.bss 段) (ELF 二进制), 317
- build process (建立过程)
 - binutils setup (binutils 的规划), 126-128
 - bootstrap compiler setup (中介编译器的规划), 128, 129
 - C library setup (C 链接库的规划), 129-133
 - compiling kernel image (编译内核映像), 172
 - configuring kernel (设定内核配置), 166-171
 - dependencies and (依赖关系与), 171, 172
 - finalizing setup (完成规划), 134-137
 - kernel headers setup (内核头部的规划), 123-125
 - overview (概述), 119
 - resources (资源), 123
 - uClibc library (uClibc 链接库), 141
- build-binutils directory (build-binutils 目录), 126
- build-glibc directory (build-glibc 目录), 129
- build-tools directory (build-tools 目录), 121, 137, 141
- bus master (总线主控者), 75
- buses (总线)
 - CompactPCI support (对 CompactPCI 的支持), 75-77
 - GPIO support (对 GPIO 的支持), 79, 80
 - I²C support (对 I²C 的支持), 80, 81
 - ISA support (对 ISA 的支持), 72
 - Linux support (Linux 所支持的), 71
 - PC/104 support (对 PC/104 的支持), 73, 74
 - PCI support (对 PCI 的支持), 72
 - PCMCIA support (对 PCMCIA 的支持), 73
 - support overview (概述所支持的), 71
 - VME support (对 VME 的支持), 74, 75
- BusyBox
 - dpkg command (dpkg 命令), 249

- embutils and (embutils 与), 197
- features (特性), 191-194
- init program (init 程序), 201-204
- module dependencies (模块的依赖关系), 175
- ping command (ping 命令), 296
- ps replacement (ps 的替代品), 333
- readelf command (readelf 命令), 184
- setup (规划), 191
- shells, 202, 251
- udhcp project (udhcp 计划), 313
- usage (用法), 192
- byte ordering (字节次序), 238, 239
- bzImage file (bzImage 文件), 173

C

- C library (C 链接库。见 diet libc; glibc; uClibc)
- C99 support (C99 的支持), 296, 304
- caching block device (带缓存的块设备), 96, 208, 214
- CAN (Controller Area Network)(控制器局域网), 107-109
- CAN in Automation (CiA) group (CiA 小组), 108, 109
- can4linux project (can4linux 计划), 108
- Canadian Crosses technique (Canadian Crosses 技术), 122
- CanFestival project (CanFestival 计划), 108
- CANopen protocol (CANopen 协议), 108
- cat command (cat 命令), 218
- CATS (computer-aided training system)(计算机辅助训练系统), 19
- CC environment variable (CC 环境变量), 129
- CF (CompactFlash)
 - booting from (引导自), 287-290
 - control module and (控制模块与), 42
 - DAQ module and (DAQ 模块与), 41
 - features (特性), 228, 229
 - JFSS2 user module and (JFSS2 用户模块与), 95
 - LILO and (LILO 与), 266-269
 - popularity of (普及), 97
- CFI (Common Flash Interface)
 - control module and (控制模块与), 42
 - functionality (功能), 214
 - kernel configuration (内核配置), 214
 - mapping drivers (mapping 驱动程序), 214
 - MTD support (对 MTD 的支持), 92, 209
 - partitioning (分区), 215-217
 - writing and reading (读写), 218, 219
- c++filt utility (c++filt 公用程序), 126
- CFLAGS variable (CFLAGS 变量) (make 命令), 138, 152
- CGI scripting (撰写 CGI 脚本), 309
- char devices (字符设备), 96, 208, 214
- chroot() system callch (root()系统调用), 269
- CHRP (Common Hardware Reference Platform), 68
- CiA (CAN in Automation) group (CiA 小组), 108, 109
- C-Kermit terminal emulator (C-Kermit 终端机仿真器), 160, 162
- clocks_in_mhz environment variable (clocks_in_mhz 环境变量), 191
- code (程序代码。见 applications; software)
- COM20020 chipset (COM20020 芯片组), 109
- COM90xx chipset (COM90xx 芯片组), 109
- Comedi package (Comedi 套件), 84, 85
- Comedilib library (Comedilib 链接库), 84
- command line (命令行)
 - IDEs, 156
 - kermit, 161, 162
- viewing kernel configuration menu (检查内核配置选项), 169
- command-daemon (gdb 服务器与), 320
- Common Flash Interface (见 CFI)
- Common Hardware Reference Platform (CHRP), 68
- communication protocols (通讯协议。见 specific protocols)
- CompactFlash (见 CF)
- CompactPCI bus (CompactPCI 总线), 75-77
- Compaq's bootldr (Compaq 的引导加载程序), 256, 259
- compilation (编译)
 - ahead-of-time compilers (预先编译器), 150
- C library cautions (C 链接库的注意事项), 132
- diet libc, 147
- Electric Fence cautions (Electric Fence 的注意事项), 340
- just-in-time compilers (实时编译器), 150
- kernel considerations (内核的考虑), 171-173
- linuxthreads package and (linuxthreads 套件与), 134
- MTD utilities (MTD 公用程序), 212
- Perl cautions (Perl 的注意事项), 151
- rsync utility (rsync 公用程序), 246

- setting up compilers (设置编译器), 128, 129, 134
 - start symbol (start 符号), 139
- telnetd, 301
- TinyLogin, 195
- U-Boot, 274-275
- uClibc library setup (uClibc 链接库的规划), 145
- udhcp, 313
- warnings during (编译) 期间的警告消息), 129
 - (另见 cross-compilation; Makefiles)
- Comprehensive Perl Archive Network (Perl 综合典藏网) (CPAN), 151, 152
- compression (压缩)
 - filesystems (文件系统), 233, 234, 242, 244
 - gzip command (gzip 命令), 243, 244
 - JFFS2, 95, 241
 - kernel configuration (内核配置), 172
- computer-aided training system(CATS)(计算机辅助训练系统), 19
- .config file (.config 文件)
 - backing up (备份), 173
 - generated by kernel (由内核产生), 166
 - kernel configuration (内核配置), 168, 169
 - multiple images (多重映像), 173
 - naming recommendations (对命名的建议), 170
 - saving manually (手动储存), 170
- CONFIG_FILTER option (CONFIG_FILTER 选项), 262
- CONFIG_FTL option (CONFIG_FTL 选项), 208
- CONFIG_MTD option (CONFIG_MTD 选项), 207
- CONFIG_MTD_BLOCK option (CONFIG_MTD_BLOCK 选项), 208
- CONFIG_MTD_BLOCK_RO option (CONFIG_MTD_BLOCK_RO 选项), 208
- CONFIG_MTD_CHAR option (CONFIG_MTD_CHAR 选项), 208
- CONFIG_MTD_PARTITIONS option (CONFIG_MTD_PARTITIONS 选项), 208
- CONFIG_NFTL option (CONFIG_NFTL 选项), 208
- CONFIG_NFTL_RW option (CONFIG_NFTL_RW 选项), 208
- CONFIG_PACKET option (CONFIG_PACKET 选项), 262
- configurability (可设定配置), 24, 38, 43
- configuration (配置)
 - add-ons (附加), 131
 - backing up (备份), 173
 - cautions enabling options (注意所启用的选项), 168
 - FreeSSH constraints (FreeSSH 的限制), 309
 - full compiler setup (完整的编译器规划), 134
 - glibc, 129-131
 - GRUB to boot from DOC (让 GRUB 从 DOC 引导), 273, 274
 - kernel-supported methods (内核所支持的方法), 168, 169
 - libc.so file (libc.so 文件), 133
 - LILO recommendations (对 LILO (配置) 的建议), 267
 - managing multiple (管理多重 (配置)), 169
 - microperl, 151
 - miniperl, 152
 - netkit-base and modifications (netkit-base 与修改), 295
 - OpenSSH, 305-306
 - processor and system type (处理器与系统类型), 125
 - rsync utility (rsync 公用程序), 246
 - saving/restoring (储存/恢复), 170
 - scripts for (脚本), 127, 129
 - setting up (设置), 45
 - uClibc, 141-145
 - unlisted features (未列出的特性), 168
 - (另见 boot configuration: dynamic configuration; kernel configuration)
- command daemon (命令服务程序)
 - DAQ module example (DAQ 模块范例), 137-140
 - diet libc, 148
 - uClibc, 146
- control module (控制模块)
 - booting from RAM disk (自 RAM disk 引导), 286-287
 - compacting (缩小尺寸), 44
 - CRAMFS and (CRAMFS 与), 235
 - embedded systems (嵌入式系统), 40, 42
 - gdb package (gdb 套件), 316, 317

- project workspace (计划工作空间), 114
 - SYSM module and (SYSM 模块与), 43
 - Controller Area Network (控制器局域网)
(CAN), 107-109
 - copyright issues (版权问题), 31-37
 - cp command (cp 命令), 185, 187, 249, 251
 - CPAN (Comprehensive Perl Archive Network)
(Perl 综合典藏网), 151, 152
 - CRAMFS filesystem (CRAMFS 文件系统)
 - automatic creation /dev entries (自动建立 /dev 目录), 189
 - compression (压缩), 234, 244
 - features (特性), 237-240
 - link count and (链接计数与), 238
 - OpenSSH and (OpenSSH 与), 308
 - RAM disks (RAM 磁盘), 243
 - selection guidelines (选用指南), 235
 - cramfsck tool (cramfsck 工具程序), 239
 - crc32 command (crc32 命令), 291
 - CROSS variable (CROSS 变量) (make 命令), 192
 - cross-compilation (交叉编译)
 - Apache and (Apache 与), 309, 312
 - considerations (考虑), 153
 - DHCP package and (DHCP 套件与), 313
 - gdb server (gdb 服务器), 317
 - libgcc constraints (libgcc 的限制), 331
 - LILO and GRUB (LILO 与 GRUB), 256
 - OpenSSH constraints (OpenSSH 的限制), 304
 - Perl and (Perl 与), 151
 - Python, 153
 - System V init program (System V init 程序), 199
 - CROSS_COMPILE variable
(CROSS_COMPILE 变量) (make 命令), 125, 171, 313
 - cross-compiling variable (cross-compiling 变量)
(make install 命令), 133
 - cryptography (密码学), 303
 - cu terminal emulator (cu 终端机仿真器), 157, 159, 160, 285
 - CUPS print management package (CUPS 打印管理套件), 90
 - CVS
 - coordinating development (协同开发), 48
 - gnat and (gnat 与), 155
 - GRUB code (GRUB 的程序代码), 257
 - retrieving code from (自 (CVS) 取回程序代码), 221
 - sh-boot and (sh-boot 与), 259
 - Cygnus (见 Red Hat)
 - Cygwin environment (Cygwin 环境) (Red Hat), 52
- ## D
- .da files (.da 文件), 332
 - daemons (服务)
 - LTT build (为 LTT 建立), 326-327
 - networking services as (网络服务启动成为), 294
 - respawning cautions (重新启动的注意事项), 308
 - xinetd as (xinetd 启动成为), 297
(另见 specific daemons)
 - DAQ (data acquisition)(资料收集)
 - booting from CF card (自 CF 卡引导), 267
 - compacting (缩小尺寸), 44
 - command daemon (命令服务), 137-140
 - disk filesystem over NFTL (NFTL 上的磁盘文件系统), 236
 - embedded system component (嵌入式系统组件), 40, 41
 - erasing DOC devices (抹除 DOC 设备), 223
 - GPIO interface and (GPIO 接口与), 80
 - hardware support (所支持的硬件), 84, 85
 - probe driver output (探测驱动程序输出), 222
 - project workspace (计划工作空间), 114
 - SYSM module and (SYSM 模块与), 43
 - .data section (.data 段) (ELF 二进制), 317
 - databases system module and (系统模块与数据库), 43, 44
 - DCS (Digital Control System)(数字控制系统), 20
 - dd command (dd 命令), 218, 243, 287
 - DDD (IDE), 322
 - Debian, 22, 69, 71
 - debugging (调试)
 - ad hoc methods (特别的方法), 48
 - BDM and JTAG interfaces (BDM 和 JTAG 接口), 56
 - filesystem recommendations (对文件系统的建议), 315
 - gdb tool (gdb 工具), 316-322
 - hardware tools (硬件工具), 342, 343

- memory debugging (内存调试), 339-342
- multibit I/O and (多输出/入与), 82
- networking interface (网络接口), 55
- performance analysis (性能分析), 330-339
- serial lines (串行线), 55
- tracing (追踪), 322-330
- U-Boot ELF binary and (U-Boot ELF 二进制文件与), 275
- virtual memory layout (虚拟内存的配置), 63
- worksheet (工作单), 351
- denial-of-access attacks (访问拒绝攻击), 296
- .depend files (.depend 文件), 171
- depmod utility (depmod 公用程序), 174
- design methodology (设计方法论), 45-49
- DESTDIR variable (DESTDIR 变量) (make 命令), 147
- /dev directory (/dev 目录), 180, 188, 189, 212
- /dev entries (/dev 项目)
 - CFI flash devices and (CFI 闪存设备与), 217
 - DOC, 223
 - LILU, 267, 269
 - MTD subsystem (MTD 子系统), 206, 207
 - worksheet (工作单), 349
- development (开发)
 - Ada, 154
 - bootloaders and (引导加载程序与), 255
 - diet libc library (diet libc 链接库), 146-148
 - differences for embedded systems (嵌入式系统的差异), 49
 - GNU Java Compiler (GNU Java 编译器), 150, 151
 - IDEs, 156, 157
 - Java, 148-151
 - Linux costs (Linux 的成本), 27
 - memory debugging cautions (内存调试的注意事项), 339
 - open source virtual machines (开放源码的虚拟机), 150
 - Perl, 151-153
 - project workspace (计划工作空间), 113-115
 - Python, 153-154
 - setting up host/target systems (设置主机/目标板系统), 52-55
 - terminal emulators (终端机仿真程序), 157-162
 - tool setup (工具程序的规划), 47, 48
 - uClibc, 140-146
 - worksheet (工作单), 345-352 (另见 GNU toolchain)
- development (framework) distributions (开发(架构)发行套件), 14
- DEVEL_PREFIX variable (DEVEL_PREFIX 变量)(uClibc), 143
- DEVEL_TOOL_PREFIX variable (DEVEL_TOOL_PREFIX 变量)(uClibc), 143
- device drivers (设备驱动程序)
 - CFI flash and (CFI 闪存与), 214
 - CIF boards (CIF 板), 111
 - DAQ vendor caveats (DAQ 厂商的注意事项), 85
 - Ethernet (以太网), 100
 - ISA bus (ISA 总线), 72
 - mapping, 209
 - PCI and (PCI 与), 72
 - SCSI interface (SCSI 接口), 78
 - self-contained MTD, 209
- DeviceNet protocol (DeviceNet 协议), 108
- dformat DOS utility (dformat DOS 公用程序), 219, 220, 224, 226, 228
- DHCP (Dynamic Host Configuration Protocol) (主机动态配置协议)
 - booting with (以 (DHCP) 引导), 282-283
 - functionality (功能), 313-314
 - GRUB network boot (GRUB 网络引导), 257
 - network boot (网络引导), 61, 261
 - setting up daemon (设置服务), 261-263
 - SYSM module and (SYSM 模块与), 43
 - UI modules and (UI 模块与), 44
- diet libc
 - embutils, 197
 - features (特性), 146-148
 - minit, 204
 - patch utility (patch 公用程序), 250
 - Python constraints (Python 的限制), 154
- diff command (diff 命令), 250, 251
- Digital Control System (DCS), 20
- dinfo command (dinfo 命令), 220, 221
- dir command (dir 命令)(gdb), 321
- directories (目录)
 - binutils, 128
 - cautioning overwriting kernels (避免覆盖原有的内核), 124
 - confusing similarities (令人困惑的相似性), 181
 - copying without GNU cp (不用 GNU cp 进行复制), 252

- downloading kernels into (将内核下载至), 123
 - GNU toolchain (GNU 工具链), 120
 - mounting on TMPFS (挂载 TMPFS 上的), 236, 244, 245
 - organizing for project (为计划组织), 113-115
 - renaming kernel directory (内核目录更名), 165
 - root filesystem (根文件系统), 179
 - setup recommendations (对规划的建议), 121
 - sharing directory trees (共享的目录树), 261
 - storage requirements (储存空间的需求), 135
 - tools directory contents (工具目录的内容), 134
 - uClibc library settings (uClibc 链接库的设置), 143
 - version numbers and (版本编号与), 170
 - disk devices (磁盘设备)
 - embedded systems (嵌入式系统), 228-231
 - LILO and (LILO 与), 266-269
 - worksheet (工作单), 347
 - disk filesystem (磁盘文件系统)
 - definition of (的定义), 236
 - GIDs and (GIDs 与), 238
 - over NFTL (NFTL 上的), 236, 241, 242
 - over RAM disk (RAM disk 上的), 242-244
 - diskboot command (diskboot 命令), 289
 - diskless systems booting (无磁盘系统的引导), 258
 - DiskOnChip (见 DOC)
 - distributions (发行套件)
 - criteria for choosing (选用的标准), 38-39
 - defined (定义), 13
 - Linux workstations (Linux 工作站), 51
 - PowerPC support (PowerPC 的支持), 67
 - survey findings (调查的结果), 22
 - target systems (目标板系统), 14
 - things to avoid (应避免之事), 39
 - using (使用), 37
 - DOC (DiskOnChip)
 - cautions using MTD utilities (注意所使用的 MTD 公用程序), 210
 - embedded system survey (嵌入式系统的调查), 22
 - features (特性), 219-228
 - functionality (功能), 214
 - GRUB and (GRUB 与), 270-274
 - JFFS2 and (JFFS2 与), 95, 236
 - LILO cautions (LILO 的注意事项), 270
 - memory device (内存设备), 91
 - MTD chip driver (MTD 芯片驱动程序), 92
 - U-Boot and (U-Boot 与), 260
 - doebbt.txt file (doebbt.txt 文件), 220, 225
 - doc_loadbios utility (doc_loadbios 公用程序: (MTD)), 210, 219, 225, 270
 - documentation (见 resources)
 - do_gettimeofday() function (do_gettimeofday() 函数), 338
 - DOS installation (DOS 的安装)
 - DOC and (DOC 与), 219, 221
 - GRUB bootloader image (GRUB 引导加载程序映像), 224
 - loadlin utility (loadlin 公用程序), 256, 258
 - DOSTATIC flag (DOSTATIC 标志), 192, 195
 - dpkg (Debian 套件), 249
 - drivers (驱动程序; 见 device drivers)
 - DSA keys (DSA 密钥), 307
 - During (正常操作期间), 58
 - dynamic configuration (动态配置), 313-314
 - Dynamic Host Configuration Protocol (见 DHCP)
 - dynamic linking (动态链接)
 - Boa, 309, 310
 - BusyBox, 191
 - copyright laws and (版权法), 33
 - gdb command (gdb 命令), 321
 - glibc package (glibc 套件), 183
 - glibc setup option (glibc 规划选项), 131, 132
 - libraries (链接库), 59
 - Python, 154
 - rsync, 246
 - shared libraries (共享链接库), 131
 - thttpd, 311
 - uClibc library (uClibc 链接库), 146
 - udhcp, 313
 - xinetd, 296
- ## E
- Eclipse (IDE), 156
 - eCos operating system (eCos 操作系统), 260
 - EDC (Evans Data Corporation), 23
 - einfo utility (einfo 公用程序)(MTD), 210
 - EISA (Extended ISA) devices (EISA (扩展 ISA) 设备), 72
 - ELC (Embedded Linux Consortium)(嵌入式 Linux 联盟), 30, 31, 363
 - ELDK (Embedded Linux Development Kit)(嵌入式 Linux 开发工具箱), 260

- Electric Fence library (Electric Fence 链接库), 340, 341
- Electromagnetic Interference (EMI)(电磁干扰), 107
- ELF binary (ELF 链接库), 275, 317, 319
- ELJ (Embedded Linux Journal), 31
- ELKS (Embeddable Linux Kernel Subset) project (ELKS 计划), 16
- Embeddable Linux Kernel Subset (ELKS) project (嵌入式 Linux 内核子集计划), 16
- embedded Linux (嵌入式 Linux), 13, 27-31
- Embedded Linux Consortium (ELC)(嵌入式 Linux 联盟), 30, 31, 363
- embedded Linux distribution (嵌入式 Linux 发行套件), 14
- Embedded Linux Journal (ELJ), 31
- embedded systems (嵌入式系统)
 - booting requirements (引导的需求), 59
 - defined (定义), 14
 - development worksheet (开发工作单), 345-352
 - examples (范例), 17-21
 - generic architecture (一般架构), 56-59
 - log file cleanup (清除日志文件), 310
 - multicomponent example (以多组件为例), 39-44
 - networking and (网络与), 17, 49
 - size determination (决定规模), 15, 16
 - survey findings (调查的结果), 21-23
 - time constraints (时间限制), 16, 17
 - ubiquity of (无所不在的), 11
 - (另见 host systems; target systems)
- Embedded Systems Programming (ESP) magazine (ESP 杂志), 21
- Emblix (Japan Embedded Linux Consortium)(日本嵌入式 Linux 联盟), 30, 363
- embutils, 197, 198
- EMI (Electromagnetic Interference)(电磁干扰), 107
- encryption (加密), 303, 320
- environment variables (环境变量)
 - automating booting process (引导过程自动化), 287
 - CC, 129
 - filesize, 284, 291
 - gdb constraints (gdb 的限制), 321
 - LD_LIBRARY_PATH, 193
 - PATH, 193, 198
 - PREFIX, 127, 143
 - saving for U-Boot (为 U-Boot 储存), 291
 - setting with script (用脚本来设定), 114
 - TARGET, 120, 127
 - U-Boot's (U-Boot 的), 277, 279
- EPROMs, 258
- erase blocks (抹除区块), 214, 217, 218
- erase command (erase 命令), 210, 218, 223, 285
- eraseall command (eraseall 命令), 210, 218, 223, 227
- erasing (抹除)
 - DOC devices (DOC 设备), 223
 - DOC install considerations (DOC 安装上的考虑), 226
 - MTD devices (MTD 设备), 241
 - U-Boot bootloader image (U-Boot 引导加载程序的映像), 291
- Ericsson blip, 19
- error messages (错误信息)
 - bad blocks (坏块), 225
 - kernel panic (内核恐慌), 209
 - partition deletion (分区删除), 228
 - SYSM module and (SYSM 模块与), 43
 - unrecognizable format (无法辨识的格式), 226
- /etc directory (/etc 目录), 180
- EtherBoot bootloader (EtherBoot 引导加载程序), 256, 258
- Ethernet (以太网)
 - 802.11 as equivalent (与 802.11 等效), 103
 - ARCnet and (ARCnet 与), 109
 - considerations using (使用上的考虑), 44
 - EMI and RFI vulnerability (EMI 及 RFI 弱点), 107
 - hardware support (所支持的硬件), 100
 - linked setup (链接式规划), 53
 - Modbus protocol and (Modbus 协议与), 110
- Eurolinux, 37
- Evans Data Corporation (EDC), 23
- exec_prefix variable (exec_prefix 变量) (make 命令), 299
- eXecute In Place (XIP)(本地执行), 95
- ext2 filesystem (ext2 文件系统)
 - data access (资料的存取), 58
 - NFTL and (NFTL 与), 96, 241
 - power-down reliability (断电可靠性), 242
 - RAM disks (RAM 磁盘), 243

ext3 journalling filesystem (ext3 日志型文件系统), 236、242

Extended ISA (EISA) devices (扩展 ISA 设备), 72

EXTRAVERSION variable (EXTRAVERSION 变量), 170、171

F

Familiar distribution (Familiar 发行套件), 154、235、249

fast infrared (FIR), 101

FAT filesystem (FAT 文件系统), 58、96、228

fcp utility (fcp 公用程序)(MTD), 210

fdisk utility (fdisk 公用程序), 98、227、228、287

FHS (Filesystem Hierarchy Standard), 30、178、181

fieldbuses (现场总线), 107-110

file command (file 命令), 135、321

files (文件)

- copying (复制), 249

- dependency considerations (依赖关系的考虑), 250

- header files (头文件), 169、171

- log file recommendations (对日志文件的建议), 310

- maps file (maps 文件), 63

- patching (补丁), 249

- transfer constraints (传输的限制), 159、275

filesize environment variable (filesize 环境变量), 284、291

Filesystem Hierarchy Standard Group, 30、364

filesystems (文件系统)

- compression (压缩), 233、234、242、244

- creating image for RAM disk (为 RAM disk 建立映像), 243

- kernel functions (内核函数), 58

- MTD utilities (MTD 公用程序), 210

- selection guidelines (选用指南), 234-236

- updating while mounted (尽管已挂载也可以更新), 245

- writing images to flash (将映像写入闪存), 237

- (另见 journalling filesystems; root filesystem)

FIR (fast infrared)(快速红外线), 101

FireWire (见 IEEE1394 standard)

firmware (固件), 220、221

flash chips (闪存芯片), 93、209、217

flash devices (闪存设备)

- blob and (blob 与), 259

- bootstrapping and (引导与), 60

- downloading binary images (下载二进制映像), 283-285

- erase blocks (抹除块), 214

- filesystems and (文件系统与), 234

- hardware worksheet (硬件工作单), 347

- JFFS2 and (JFFS2 与), 236

- JTAG dongles (JTAG 转换器), 343

- RAM location and (RAM 位置与), 275

- system memory layout (系统内存的配置), 63

- writing filesystem image (写入文件系统映像), 237

- (另见 specific models)

Flash Translation Layer(FTL)(快速瞬态装入程序), 96、208、211

floppy disks (软盘), 231

foreign code licensing clarifications (对外部程序代码的版权声明), 370-375

free() function (free()函数), 340、341

free software community (自由软件社区 见 open source)

Free Software Foundation (FSF)(自由软件基金会), 11、30、364

Free Standards Group (FSG)(自由标准小组), 30、364

FreeIO (Free Hardware Resources for the Free Software Community)(自由软件社群的自由硬件资源), 364

Freshmeat web site (Freshmeat 网站), 26

FreSSH package (FreSSH 套件), 309

FTL (Flash Translation Layer)(快速瞬态装入程序), 96、208、211

ftl_check utility (MTD)(ftl_check 公用程序), 211

ftl_format utility (MTD)(ftl_format 公用程序), 211

G

garbage collection (垃圾收集), 240

gasp utility (gasp 公用程序)(binutils), 126

gcc (GNU C compiler)(GNU C 编译器)

- binary utility cautions (二进制公用程序的注意事项), 136

- code coverage recommendations (对程序代码涵盖范围的建议), 331

- cross-compilation and (交叉编译与), 153

- debugging options (调试选项), 319

- extracting (取出), 128
- gcc component (gcc 组件), 128
- gdb and (gdb 与), 317
- gnat constraints (gnat 的限制), 155
- GPL license (GPL 授权书), 32
- installing (安装), 129
- resources (资源), 122
- setting up (设置), 128, 129
- versions (版本), 116, 118
- gcj (见 GNU Java Compiler)
- gcov utility (gcov 公用程序), 48, 331, 332
- gdb (GNU debugger)(GNU 调试器)
 - BDM and JTAG debuggers (BDM 与 JTAG 调试器), 56
 - building/installing components (建立/安装组件), 316-317
 - debugging applications (对应用程序进行调试), 317-321
 - Electric Fence, 341
 - GPL license (GPL 授权书), 32
 - popularity of (的普及), 48
- gdb stubs, 316
- .gdbinit file (.gdbinit 文件), 321
- General-Purpose Interface Bus (GPIB)(通用接口总线), 80, 81
- get_cycles() function (get_cycles()函数), 338
- gettimeofday(), 246
- GhostScript package (GhostScript 套件), 90
- GID field (GID 字段), 238
- glib package (glib 套件), 308
- glibc (GNU C library)(GNU C 链接库)
 - alternatives (替代品), 140
 - applications and (应用程序与), 59
 - Boa, 309, 310
 - build considerations (建立时的考虑), 129-131
 - BusyBox and (BusyBox 与), 192
 - compilation cautions (编译时的注意事项), 132
 - components in (其中的组件), 183-184
 - delicacy of build (建立过程最麻烦), 129
 - FPU emulation option (GNU 仿真选项), 132
 - FreSSH, 309
 - GNU toolchain version combinations (GNU 工具链的版本组合), 117
 - inetd support (对 inetd 的支持), 295
 - JDK and JRE (JDK 与 JRE), 149
 - kernel headers and (内核头部与), 118
 - LGPL license (LGPL 授权书), 32
 - library build options (链接库建立选项), 131
 - linking options (链接选项), 131, 132
 - linuxthreads, 129, 132
 - microperl and (microperl 与), 152
 - Net-SNMP and (Net-SNMP 与), 298, 299
 - OpenSSH, 304
 - package download (套件下载), 115
 - patch utility (patch 公用程序), 250
 - Python considerations (Python 的考虑), 154
 - resources (资源), 122
 - root filesystem (根文件系统), 182-186
 - rsync utility (rsync 公用程序), 246
 - setting up (设置), 129-133
 - shell script updating (用 shell 脚本更新), 252
 - strace and (strace 与), 323
 - telnetd, 301, 302
 - thttpd, 311
 - TinyLogin, 194, 195
 - trace daemon and (trace 服务与), 326
 - udhcp and (udhcp*与), 313
 - Unix systems and (Unix 系统与), 52
 - utelnetsd, 302
 - version considerations (版本的考虑), 116, 118
 - worksheet (工作单), 349
 - xinetd support (对 xinetd 的支持), 296
- glibc-encrypt, 115
- glibc-linuxthreads, 118
- Glimmer (IDE), 156
- gnat (GNU Ada) compiler (gnat 编译器), 155
- GNU C library (见 C library)
- GNU GPL (General Public License)(通用公共授权书)
 - binary kernel modules (二进制内核模块), 366-370
 - Boa and (Boa 与), 309
 - BusyBox package (BusyBox 套件), 191
 - "contamination" (污染), 33, 34
 - diet libc licensing (diet libc 的授权), 146
 - FSF and (FSF 与), 30
 - kernel license exclusion (排除在内核版权之外), 365
 - licensing (授权), 32-34
 - Linux code availability (Linux 程序代码的可用性), 24
 - M-Systems DOC driver (M-Systems DOC 驱动程序), 223
 - RTLinux patent and (RTLinux 专利与), 36

- udhep project (udhep 计划), 313
- utelnctd, 302
- GNU Java Compiler (GNU Java 编译器)(gjc), 150, 151
- GNU toolchain (GNU 工具链)
 - BDM/JTAG debuggers and (BDM/JTAG 调试器与), 56
 - binutils setup (binutils 的规划), 126-128
 - build overview (建立程序概述), 119
 - build-tools directory (build-tools 目录), 121
 - C library setup (C 链接库的规划), 129-133
 - component versions (组件的版本), 116-118
 - Cygwin environment (Cygwin 环境), 52
 - finalizing setup (完成规划), 134-137
 - gcc setup (gcc 的规划), 122, 123, 128, 129
 - kernel headers setup (内核标头的规划), 123-125
 - memory for compilation (编译时所需的内存空间), 51
 - overview (概述), 115
 - resources (资源), 122
 - sharing tools (共享工具程序), 121
 - Unix systems and (Unix 系统与), 52
 - using (使用), 137-140
 - workspace setup (工作空间的规划), 120-121 (另见 build process)
- GNUPro product (GNUPro 产品)(Red Hat), 156
- GOAS (Ground Operator Assistant System)(地面操作员辅助系统), 21
- GPIO (General-Purpose Interface Bus)(通用接口总线), 80, 81
- GPL (见 GNU GPL)
- gprof utility (gprof 公用程序), 48, 330, 331, 335
- graphical interface (图形接口)(X Window System), 89
- GRUB (GRand Unified Bootloader)
 - bootloader image (引导加载程序的映像), 224
 - comparison (比较), 256
 - cross-compilation (交叉编译), 256
 - DOC devices and (DOC 设备与), 270-274
 - features (特性), 257
 - version considerations (版本的考虑), 226
- GTK widget toolkit (GTK 窗口组件工具集), 32, 327
- gzip command (gzip 命令), 172, 243, 244

H

- hard real-time system (硬实时系统), 16, 42
- hardware support (所支持的硬件)
 - ARM processor (ARM 处理器), 66, 67
 - buses and interfaces (总线与接口), 71-81
 - debugging tools (调试工具), 56, 342, 343
 - IBM/Motorola PowerPC, 67, 68
 - industrial grade networking (工业级网络能力), 107-111
 - input/output (输入/输出), 81-91
 - kernel configuration options (内核配置选项), 167
 - kernprof and (kernprof 与), 335
 - Linux and (Linux 与), 25, 36, 51
 - MIPS processor (MIPS 处理器), 68-69
 - Motorola 68000, 71
 - networking (网络), 100-107
 - open source projects (开放源码计划), 364
 - processor architectures (处理器架构), 64
 - storage (存储设备), 91-99
 - SuperH, 70
 - U-Boot, 260
 - x86 processor (x86 处理器), 65, 66
- HCI (Host Controller Interface), 105, 106
- heidump tool (heidump 工具程序)(BlueZ), 106
- hdparm utility (hdparm 公用程序), 98
- header files (头文件), 169, 171
- help command (help 命令), 288
- HelpPC shareware (HelpPC 共享软件), 66
- hexadecimal format (十六进制格式), 277
- HEYU! project (HEYU! 计划), 87
- high-availability (高可用), 76
- Hitachi SuperH (见 SuperH processor)
- home automation (家庭自动化), 86, 87
- /home directory (/home 目录), 179
- Host Controller Interface (HCI), 105, 106
- host systems (主机系统)
 - automatic network configuration (自动设定网络配置), 313
 - byte ordering considerations (字节顺序的考虑), 238, 239
 - debug setups (调试规划), 55, 56
 - defined (的定义), 14
 - development setups (开发规划), 52-55
 - GNU toolchain (GNU 工具链), 116, 117
 - installing MTD utilities (安装 MTD 公用程序), 211-212

- testing connections (测试连接), 48
- types of (类型), 50-52
- Hot Swap specification (热插拔规范)
 - (CompactPCI), 75
- HTTP
 - SYSM module and (SYSM 模块与), 43
 - web content and (web 内容与), 309-312
- I
- I²C (Inter-Integrated Circuit) bus (I²C 总线),
 - 80, 81
- i386 platform (i386 平台)
 - embutils, 197
 - GNU toolchain (GNU 工具链), 117
 - hardware support (所支持的硬件), 65
 - PCMCIA support (对 PCMCIA 的支持), 73
 - target build example (目标板建立范例), 118
- i386-linux directory (i386-linux 目录), 135
- IAP (Information Access Protocol)(信息存取协议), 102
- IAS (Information Access Service)(信息存取服务), 100
- IBM/Motorola PowerPC (见 PowerPC)
- ICE (In-Circuit Emulator)(在线仿真器), 56, 343
- ide commands (ide 命令), 288, 290
- IDE drives (IDE 磁盘驱动器)(另见 ATA-ATAPI)
 - CF cards and (CF 卡与), 228
 - CompactFlash devices as (把 CompactFlash 设备做为), 95
 - LILO and (LILO 与), 267, 268
 - U-Boot and (U-Boot 与), 260
- IDEs (integrated development environments)(集成开发环境)
 - availability (可用性), 157
 - listed (列出), 156
 - using (使用), 48
 - worksheet (工作单), 347
- IEEE1284 standard (IEEE1284 标准), 77
- IEEE1394 (Firewire) standard (Firewire 标准),
 - 79, 80, 167
- IEEE488 (GPIB) standard (GPIB 标准), 80
- IEEE 802.11 (wireless) standard (无线标准),
 - 103, 104
- IETF standard (IETF 标准), 303
- iminfo command (iminfo 命令), 282, 284
- implementation methodology (实现方法论),
 - 45-49
- In-Circuit Emulator (ICE)(在线仿真器), 56, 343
- include directory (include 目录), 125, 135
- index.html files (index.html 文件), 311
- Industry Standard Architecture (ISA), 72, 74
- inetd super-server (inetd 超级服务器), 264, 294-296, 302
- info directory (info 目录), 135
- Infrared Data Association (见 IrDA)
- init program (init 程序)
 - BusyBox init, 201-204
 - kernel and (kernel 与), 198
 - Minit, 204
 - respawning cautions (重新启动的注意事项), 308
 - standard System V init (标准的 System V init), 199, 201
 - start_kernel() function (start_kernel()函数), 59
 - system startup component (系统启动组件), 59
- Initial Program Loader (IPL), 223, 270
- initialization (见 system initialization)
- initramfs (init RAMFS), 245
- initrd mechanism (initrd 机制), 242, 245
- installation (安装)
 - bootloader image (引导加载程序的映像), 223, 224
 - C library (C 链接库), 132
 - checking for binutils (检视 binutils 的), 127
 - distribution considerations (发行套件的考虑), 38
 - DOS method for DOC (以 DOS 的方法(安装)DOC), 219
 - embutils, 197
 - full compiler (完整的编译器), 134
 - gcc, 129
 - gdb, 316-317
 - GRUB on DOC (将 GRUB (安装)在 DOC), 273
 - inetd, 295
 - kernel considerations (内核考虑), 173-175
 - MTD utilities (MTD 的公用程序), 211-214
 - patch utility (patch 公用程序), 250
 - rsync utility (rsync 公用程序), 246
 - strace tool (strace 工具程序), 323
 - U-Boot, 274-275
 - uClibc library (uClibc 链接库), 145
 - udhcp, 313, 314

- visualization tool (可视化工具), 327
 - (另见 DOS installation)
 - INSTALL_MOD_PATH variable
 - (INSTALL_MOD_PATH 变量)
 - (make 命令), 174
 - install_root variable (install_root 变量) (make install 命令), 132
 - INT 18h, 271
 - INT 19h, 270
 - Intel (见 x86 processors)
 - Interbus fieldbus, 111
 - interfaces (接口)
 - Ada, 155
 - CompactFlash access via (CompactFlash 存取经由), 97
 - DAQ hardware (DAQ 硬件), 84
 - hardware support (所支持的硬件), 77-80
 - International Space Station (ISS)(国际太空站), 20
 - Internet Software Consortium (ISC)(因特网软件联盟), 313
 - internet super-servers (internet 超级服务器)
 - DHCP and (DHCP 与), 261
 - enabling TFTP service (启用 TFTP 服务), 264
 - inetd, 294-296
 - special daemon (特殊的服务), 294
 - xinetd, 296, 297
 - interpreters (解释器)
 - microperl, 151
 - miniperl, 152
 - Perl, 151
 - Python, 153
 - interrupt handlers (中断处理例程), 270, 338
 - interrupt latency (中断等待时间), 337-339, 343
 - intrusions NFS service and (侵入、NFS 服务与), 265
 - I/O (input/output)(输出/入)
 - generic requirements (一般需求), 57
 - hardware support (所支持的硬件), 81-91
 - logic analyzers (逻辑分析仪), 343
 - IP addresses (IP 地址)
 - automatic configuration and (自动设定配置与), 313
 - control module and (控制模块与), 42
 - DAQ modules and (DAQ 模块与), 41
 - SYSM module and (SYSM 模块与), 43
 - UI modules and (UI 模块与), 44
 - iPKG (Itsy Package Management System), 249
 - IPL (Initial Program Loader), 223, 270
 - IrCOMM layer (IrCOMM 层), 102
 - IrDA (Infrared Data Association)(红外线数据协会)
 - Bluetooth and (蓝牙与), 105
 - hardware support (所支持的硬件), 100-103
 - kernel support options (内核支持选项), 167
 - IrLAN, 103
 - IrLAP (link access protocol)(链接存取协议), 100, 102
 - IrLMP (link management protocol)(链接管理协议), 100, 102
 - IrNET, 103
 - IrOBEX, 103
 - IrPHY (physical signaling layer)(物理信号层), 100
 - IrPORT driver (IrPORT 驱动程序)(IrDA), 102
 - IrTTY driver (IrTTY 驱动程序)(IrDA), 102
 - ISA (Industry Standard Architecture), 72, 74
 - ISC (Internet Software Consortium)(Internet 软件联盟), 313
 - ISO 11898 standard (ISO 11898 标准)(CAN), 108
 - ISS (International Space Station)(国际太空站), 20
 - Itsy Package Management System (iPKG), 249
- ## J
- J1939 protocol (J1939 协议), 108
 - Japhar project (Japhar 计划), 150
 - Java Development Kit (JDK)(Java 开发工具集), 149
 - Java programming language (Java 程序语言)
 - ALICE project (ALICE 计划), 87
 - background (背景), 148-151
 - Blackdown project (Blackdown 计划), 66
 - Motorola 68000 processors and (Motorola 68000 处理器与), 71
 - PowerPC support (对 PowerPC 的支持), 67
 - SuperH processors and (SuperH 处理器与), 70
 - Java Runtime Environment (JRE)(Java 运行时环境), 148, 149
 - Java Virtual Machine (JVM)(Java 虚拟机), 148, 150
 - JDK (Java Development Kit)(Java 开发工具), 149
 - JEDEC Solid State Technology Association (JEDEC 固态技术协会), 93

- JFFS (MTD), 209
- JFFS2 filesystem (JFFS2 文件系统;
 - automatic creation /dev entries (自动建立 /dev 目录), 189
 - blob and (blob 与), 259
 - compression (压缩), 234, 244
 - DOC and (DOC 与), 95, 236
 - erase blocks and (抹除区块与), 218
 - features (特性), 94, 240, 241
 - MTD support (对 MTD 的支持), 209
 - selection guidelines (选用指南), 235
 - storage support (对储存设备的支持), 58
 - U-Boot and (U-Boot 与), 260
- jffs2reader utility (jffs2reader 公用程序) (MTD), 210
- JFS jouralling filesystem (JFS 日志型文件系统), 242
- jModbus project (jModbus 计划), 110
- joeq VM project (joeq VM 计划), 150
- jouralling filesystems (日志型文件系统)
 - documentation (文件), 237
 - JFFS2 and (JFFS2 与), 94
 - NFTL and (NFTL 与), 96, 242
 - power-down reliability and (断电可靠性与), 236
- Jouralling Flash File System (JFFS) user module (JFFS 用户模块), 96
- JRE (Java Runtime Environment)(Java 运行时环境), 148, 149
- JTAG debugger (JTAG 调试器), 48, 56, 343
- just-in-time (JIT) compilers (实时编译器), 150
- JVM (Java Virtual Machine)(Java 虚拟机), 148, 150
- K**
- Kcomedilib, 84
- KDevelop (IDE), 156, 322
- keepalive signals (keepalive 信号), 44
- kermit utility (kermit 公用程序), 157, 161, 284
- .kermrc configuration file (.kermrc 配置档), 161
- kernel (内核)
 - ALSA integration, 89
 - Appicom cards, 111
 - architecture name selection (选择架构名称), 167
 - ARCnet support (对 ARCnet 的支持), 109
 - ATA/IDE support (对 ATA/IDE 的支持), 98
 - binary modules notices (二进制模块注意事项), 366-370
 - blob and (blob 与), 259
 - bootstrapping requirements (引导的需求), 59
 - building (建立), 46, 47
 - CFI specification support (对 CFI 规格的支持), 92
 - compiling (编译), 171-173
 - dealing with failure (故障处理), 175-177
 - debugging (调试), 48, 55, 70
 - display support (对显示的支持), 89
 - DOC driver (DOC 驱动程序), 223
 - documentation (文件), 163
 - embedded Linux and (嵌入式 Linux 与), 13
 - failure to boot after updates (更新之后无法引导), 292
 - filesystem engines (文件系统引擎), 58
 - functions of (的特性), 12
 - generic requirements (一般需求), 57
 - GNU toolchain and (GNU 工具链与), 117
 - GPL, 32, 34
 - I²C, 81
 - importance of (的重要性), 163
 - initrd images (initrd 映像), 242
 - installing (安装), 173-175, 187
 - I/O device support (对输出设备的支持), 81
 - Kcomedilib, 84
 - layered services (分层的服务), 58
 - legal clarifications (澄清授权范围), 370-375
 - LTT and (LTT 与), 28, 325, 326
 - MontaVista contributions (MontaVista 发行套件), 29
 - Motorola 68000 processors (Motorola 68000 处理器), 71
 - MTD and (MTD 与), 91, 205, 206
 - OS functions (操作系统的特性), 57
 - pointer devices and (指位设备与), 88
 - /proc filesystem (/proc 文件系统), 333
 - RAM and (RAM 与), 242, 284
 - renaming directory (目录更名), 165
 - root filesystem requirements (根文件系统的需求), 58
 - SCSI layer (SCSI 层), 78
 - secondary kernels under (之下的第二个内核), 14
 - selecting (选择), 46, 163-166

- system startup component (系统启动组件), 59
 - TrueFFS tools (TrueFFS 工具程序), 219
 - USB and (USB 与), 79
 - version variations (版本编号的变化), 165
 - virtual address space (虚拟地址空间), 62
 - watchdog timers (watchdog 定时器), 112
 - worksheet (工作单), 348
 - kernel configuration (内核的配置)
 - building and (建立与), 45, 124
 - CHI flash, 214
 - considerations (考虑), 166-171
 - DOC, 221
 - kernel selection and (内核的选用与), 46
 - MIPS, 69
 - MTD subsystem (MTD 子系统), 207-209
 - rebuilding toolchain and (重建工具链与), 125
 - kernel headers (内核的头部)
 - build requirement (建立的需求), 118
 - configuring (设定配置), 131
 - setup (规划), 119, 123-125
 - kernel panic (内核恐慌)
 - code location (程序代码的位置), 176
 - example (范例), 283
 - MTD and (MTD 与), 209
 - premature exit and (提前结束并且), 198
 - reasons for (原因), 175
 - sample process (实例), 177
 - system reboot and (系统重新引导与), 176
 - kernel profiling (内核统计), 335-337
 - Kernel Traffic newsletter (Kernel Traffic 周报), 165
 - KERNEL_SOURCE variable (KERNEL_SOURCE 变量) (uClibc), 143
 - kernprof tool (kernprof 工具), 335
 - KERN_WARNING symbol (KERN_WARNING 符号), 212, 213
 - keyboards (键盘), 87
 - keys (密钥), 303, 307
 - Kissme project (Kissme 计划), 150
 - Kurt project (Kurt 计划), 14
- L**
- L2CAP (Logical Link Control and Adaptation Protocol), 106
 - l2ping tool (BlueZ), 106
 - LART project (LART 计划), 259, 343, 364
 - ld utility (ld 公用程序), 126, 135, 186
 - ldd command (ldd 命令), 184
 - LDFLAGS option (LDFLAGS 选项) (make 命令)
 - linking option (链接选项), 309, 311
 - Makefile example (Makefile 范例), 138
 - static linking (静态链接), 326
 - strace, 323
 - udhcp, 313
 - LD_LIBRARY_PATH environment variable (LD_LIBRARY_PATH 环境变量), 193
 - LDPS (Linux Development Platform Specification), 30
 - LDSHARED variable (LDSHARED 变量) (configure), 213
 - 1.GPL, 30, 32-34
 - /lib directory (/lib 目录), 135, 185, 186
 - libcrypt (cryptography library)(密码链接库), 186, 296, 311
 - libc.so file (libc.so 文件), 133, 134
 - libdl (dynamic loading library)(动态加载链接库), 186, 298
 - libgcc (gcc 链接库), 331
 - libm (math 链接库), 186, 296, 298
 - libraries (链接库)
 - file dependencies and (文件依赖关系与), 250
 - installing on root filesystem (安装在根文件系统), 181-187
 - LGPL and (1.GPL 与), 32
 - linking of (的链接), 59
 - location of shared (共享(链接库)的位置), 131
 - stripping (除去符号表), 185 (另见 system libraries)
 - libutil (login routines library), 186, 302
 - licensing (授权)
 - Apache, 312
 - Blackdown project (Blackdown 计划), 149
 - C-Kermit, 161
 - diet libc and (diet libc 与), 146
 - distribution considerations (发行套件的考虑), 38
 - exclusion of user-space applications (排除用户空间的应用程序), 365
 - GPL and LGPL (GPL 与 LGPL), 32-34
 - inetd, 294
 - kernel, 370-375
 - Linux and (Linux 与), 26
 - Net-SNMP, 298

- OpenSSH, 303
 - thttpd, 309
 - xinetd, 296
 - (另见 BSD license; GNU GPL)
 - Lilo (Linux LOader) bootloader (LILO 引导加载程序), 256, 266-269
 - linear variable differential transformers (LVDTs) (线性差动变压器), 40
 - Lineo survey findings (Lineo 的调查结果), 22
 - linking (链接)
 - applications to C library (应用程序 (链接) C 链接库), 133
 - considerations for libraries (考虑到链接库的), 59
 - diet libc with applications (diet libc 与应用程序), 148
 - miniperl and (miniperl 与), 152
 - proprietary applications and (私有应用程序与), 33
 - uClibc library and applications (uClibc 链接库与应用程序), 146
 - (另见 dynamic linking; static linking)
 - Linux, 12, 23-27
 - Linux Development Platform Specification (LDPS)(Linux 开发平台规范), 30
 - Linux distributions (Linux 发行套件。见 distributions)
 - Linux From Scratch project (Linux From Scratch 计划), 190
 - Linux Journal
 - accelerator control example (以加速器控制为例), 18
 - CATS example (以 CATS 为例), 19
 - resource (资源), 31
 - SCADA protocol converter example (以 SCADA 协议转换器为例), 20
 - space vehicle control example (以太空载具控制为例), 20
 - Linux kernel (Linux 内核。见 kernel)
 - Linux Standard Base (LSB), 30, 364
 - Linux systems (Linux 系统。见 systems)
 - Linux Trace Toolkit (见 LTT)
 - Linux workstations (Linux 工作站), 50, 51
 - LinuxBIOS bootloader (LinuxBIOS 引导加载程序), 256, 258
 - LinuxDevices.com, 22, 31
 - LinuxPPC support (对 LinuxPPC 的支持), 67
 - Linux/RK project (Linux/RK 计划), 14
 - linuxthreads package (linuxthreads 套件)
 - compiler cautions (编译器的注意事项), 134
 - compiling glibc without (编译 glibc 不需要), 132
 - configuring (设定配置), 131
 - library setup (链接库规划), 129
 - live updates (动态更新), 245-252
 - LMbench tool (LMbench 工具程序), 335
 - lm_sensors package (lm_sensors 套件), 112
 - loadb command (loadb 命令), 284
 - loadlin utility (loadlin 公用程序)(DOS), 256, 258
 - loads command (loads 命令), 285
 - lock utility (lock 公用程序)(MTD), 210
 - log files (日志文件), 310
 - logic analyzers (逻辑分析仪), 343
 - logical address (逻辑地址)(见 virtual address)
 - Logical Link Control and Adaptation Protocol (L2CAP), 106
 - LonWorks fieldbus, 111
 - loopback constraints (环回的限制), 241
 - LPD print management package (LPD 打印管理套件), 90
 - LPRng print management package (LPRng 打印管理套件), 90
 - LSB (Linux Standard Base), 30, 364
 - LSH (SSH implementation), 308
 - LTT (Linux Trace Toolkit)
 - authorship (原作者), 28
 - building trace daemon (建立追踪服务), 326-327
 - features (特性), 323-325
 - MontaVista contributions (MontaVista 的贡献), 29
 - patching the kernel (为内核打补丁), 325, 326
 - tracing target (追踪目标), 328-330
 - visualization tool (可视化工具), 327
 - LVDTs (linear variable differential transformers) (线性差动变压器), 40
 - LynuxWorks, 29
- ## M
- M68k processors (M68k 处理器)
 - appropriate kernel location (最适合的内核供应网站), 164
 - architecture overview (架构概述), 71
 - bootloader availability (引导加载程序的可用性), 255

- bootloader comparison (引导加载程序的比较), 256
- kernel architecture name (内核架构名称), 167
- RedBoot, 257
- UI modules and (UI 模块与), 44
- Machine Automation Tools LinuxPLC (MAT LPLC), 85, 110
- main() function (main() 函数), 139
- make clean command (make clean 命令), 132, 212
- make command (make 命令)
 - ARCH variable (ARCH 变量), 125, 167, 171
 - CFLAGS variable (CFLAGS 变量), 138, 152
 - CROSS variable (CROSS 变量), 192
 - CROSS_COMPILE variable (CROSS_COMPILE 变量), 125, 171, 313
 - DESTDIR variable (DESTDIR 变量), 147
 - INSTALL_MOD_PATH variable (INSTALL_MOD_PATH 变量), 174
 - OpenSSH considerations (OpenSSH 的考虑), 307
 - prefix and exec_prefix variables (prefix 与 exec_prefix 变量), 299
 - PREFIX variable (PREFIX 变量), 120, 192, 299
 - static linking (静态链接), 309, 311
 - TARGET_ARCH variable (TARGET_ARCH 变量), 192 (另见 LDFLAGS 选项)
- make config command (make config 命令), 168, 169
- make distclean command (make distclean 命令), 212
- make install command (make install 命令)
 - C library assumptions (C 链接库会认为), 133
 - inetd cautions (inetd 的注意事项), 295
 - install-root variable (install-root 变量), 132
 - telnetd cautions (telnetd 的注意事项), 301
- make menuconfig command (make menuconfig 命令), 169, 170
- make oldconfig command (make oldconfig 命令), 168, 170
- make xconfig command (make xconfig 命令), 169, 170
- MAKEDEV script (MAKEDEV 脚本)(MTD), 212, 213
- Makefiles
 - building compiler (建立编译器), 129
 - building dependencies (建立依赖关系), 171
 - BusyBox configuration (BusyBox 的配置), 192
 - controlling creation of (控制 (Makefile) 的产出), 127
 - DHCP and cross-compilation (DHCP 与交叉编译), 313
 - diet libc compilation (diet libc 的编译), 147
 - example (范例), 137-140
 - gcc modifications for code coverage (针对程序代码涵盖范围对 gcc 进行修改), 332
 - installing MTD utilities (安装 MTD 公用程序), 212
 - modifying for process profiling (针对进程统计进行修改), 330
 - modules_install target (modules_install 建立目标), 174
 - System V init program (System V init 程序), 200
 - vmlinux target (vmlinux 建立目标), 172
 - zImage target (zImage 建立目标), 172 (另见 compilation)
- malloc() function (malloc() 函数), 340, 341
- man directory (man 目录), 135
- mapping drivers (mapping 驱动程序), 94, 209, 214
- maps file (maps 文件), 63
- MAT LPLC (Machine Automation Tools LinuxPLC), 85, 110
- measuring interrupt latency (测量中断等待时间), 337
- medium speed infrared (中速红外线)(MIR), 101
- memory (内存)
 - C library compilation and (C 链接库编译与), 132
 - debugging (调试), 339-342
 - kernel functions (内核的特性), 58
 - layout considerations (配置上的考虑), 62-63
 - Linux workstations (Linux 工作站), 51
 - memory devices (内存设备), 91
 - physical memory map (物理内存配置图), 62
 - swapping (置换), 231 (另见 RAM)

- memory management unit (内存管理单元。见 MMU)
 - memory technology device (内存技术设备。见 MTD)
 - Memory Technology Device Subsystem project (内存技术设备子系统计划), 97
 - MEMWATCH library (MEMWATCH 链接库), 341-342
 - messaging (传讯) (Modbus 的两种格式), 110
 - metadata compression and (元数据压缩与), 234
 - microperl build option (microperl 建立选项), 151, 152
 - Microwindows, 29
 - mild time constraints (宽容时限), 17
 - minicom terminal emulator (minicom 终端机仿真程序), 157, 159, 275
 - miniperl build option (miniperl 建立选项), 151, 152, 153
 - Minit program (Minit 程序), 204
 - MIPS processor (MIPS 处理器)
 - architecture overview (架构概述), 68-69
 - bootloader (引导加载程序), 255, 256
 - diet libc support (对 diet libc 的支持), 146
 - embutils, 197
 - GNU toolchain (GNU 工具链), 117
 - kernel and (内核与), 164, 167
 - PMON and (TrueFFS 与), 257, 259
 - U-Boot and (U-Boot 与), 257
 - UI modules and (UI 模块与), 44
 - MIR (medium speed infrared)(中速红外线), 101
 - MisterHouse project (MisterHouse 计划), 86
 - mkcramfs tool (mkcramfs 工具), 239
 - mke2fs command (mke2fs 命令), 243, 244
 - mkfs.jffs utility (mkfs.jffs 公用程序)(MTD), 210
 - mkfs.jffs2 utility (mkfs.jffs2 公用程序)(MTD), 210, 240
 - mkimage utility (mkimage 公用程序), 280, 281
 - mknod command (mknod 命令), 189, 212
 - MMU (memory management unit)(内存管理单元), 57, 64, 70, 71
 - /mnt directory (/mnt 目录), 179
 - Modbus protocol (Modbus 协议), 110
 - modems hardware support (调制解调器所支持的硬件), 83, 84
 - modprobe docprobe command (modprobe docprobe 命令), 222
 - modules_install target (modules_install 建立目标), 174
 - monitoring systems (监控系统), 111, 112
 - monitors bootloaders and (监控程序引导加载程序与), 255, 256
 - MontaVista, 22, 29
 - Motorola (见 M68k processors; PowerPC)
 - mounting (挂载)
 - constraints using loopback (使用 loopback 的限制), 241
 - directories on TMPFS (TMPFS 上的目录), 236, 244, 245
 - filesystem considerations (文件系统的考虑), 234
 - JFFS2 filesystem (JFFS2 文件系统), 241
 - partitions (分区), 242
 - root filesystem (根文件系统), 209
 - mouse hardware support (鼠标所支持的硬件), 88
 - M-Systems, 96, 219, 223
 - (另见 DOC; loadlin utility)
 - MTD (memory technology device)(内存技术设备)
 - blob and (blob 与), 259
 - chip drivers (芯片驱动程序), 92
 - DiskOnChip and (DiskOnChip 与), 219-228
 - filesystems and (文件系统与), 236
 - hardware support (所支持的硬件), 91-97
 - installing utilities (安装公用程序), 211-214
 - kernel and (内核与), 205
 - Native CFI Flash and (原生的 CFI 闪存与), 214-219
 - reprogramming boot storage media (再次烧录引导储存媒体), 60
 - usage basics (基本用法), 206-214
 - writing JFFS2 to (将 JFFS2 写入), 241
 - mtd_debug utility (mtd_debug 公用程序)(MTD), 210
 - mtd_info structure (mtd_info 结构), 92
 - multibit I/O (多输出/入), 82
 - MyLinux project (MyLinux 计划), 364
- ## N
- Name Switch Service (见 NSS)
 - naming conventions (命名习惯), 137, 170
 - NAND flash (NAND 闪存)
 - DOC and (DOC 与), 93
 - functionality (功能), 214

- JFFS2 and (JFFS2 与), 95, 236
 - MTD support (对 MTD 的支持), 209, 211
 - NAND Flash Translation Layer (见 NFTL)
 - nanddump utility (nanddump 公用程序)(MTD), 211
 - nandtest utility (nandtest 公用程序)(MTD), 211
 - nandwrite utility (nandwrite 公用程序)(MTD), 211
 - nanokernels (超微内核), 36
 - netkit package (netkit 套件), 294, 300
 - netkit-base package (netkit-base 套件), 294
 - netkit-rsh package (netkit-rsh 套件), 246
 - netkit-telnet package (netkit-telnet 套件), 301
 - Net-SNMP package (Net-SNMP 套件), 298
 - network adapters (网络适配卡), 43
 - network boot (网络引导), 61, 257, 261-265
 - Network Interface Card (网络适配卡)(NIC), 109
 - network login (网络登录), 297, 300-303
 - networks (网络)
 - debugging using (调试使用), 55
 - dynamic configuration (动态配置), 313-314
 - embedded systems and (嵌入式系统与), 17, 49, 293
 - fieldbuses (现场总线), 107-110
 - hardware support (所支持的硬件), 100-107
 - industrial grade (工业等级), 107-111
 - internet super-servers (internet 超级服务器), 294-297
 - kernel functions and protocols (内核的特性与协议), 58
 - remote administration (远程管理), 297-300
 - secure communication (安全通讯), 303-309
 - web content and HTTP (web 内容与 HTTP), 309-312
 - worksheet (工作单), 350
 - NFS
 - booting with (引导使用), 282-283
 - debugging recommendations (对调试的建议), 315
 - mounting root filesystem (挂载跟文件系统), 265-266
 - network boot (网络引导), 261
 - tracing (追踪), 328
 - writing to flash (写入闪存), 237
 - NFTL (NAND Flash Translation Layer)
 - disk filesystem (磁盘文件系统), 236, 241, 242
 - DOC devices (DOC 设备), 219, 223, 224-227
 - features (特性), 96
 - journalling filesystems (日志型文件系统), 242
 - MTD support (对 MTD 的支持), 208, 211
 - nftldump utility (nftldump 公用程序)(MTD), 211
 - nftl_format command (nftl_format 命令), 211, 219, 224, 225, 226, 227
 - NIC (Network Interface Card)(网络接口卡), 109
 - nm utility (nm 公用程序)(binutils), 126, 135
 - NOR flash devices (NOR 闪存), 214
 - notifier_chain_register function (notifier_chain_register 函数), 176
 - notifier_chain_unregister function (notifier_chain_unregister 函数), 176
 - NSS (Name Service Switch)(名称服务切换)
 - glibc, 131, 132, 185
 - strace, 323
 - TinyLogin, 195
 - udhcp, 313
- ## O
- OBEX, 105
 - objcopy utility (objcopy 公用程序)(binutils), 126
 - objdump utility (objdump 公用程序)(binutils), 126, 317
 - Ocan driver project (Ocan 驱动程序计划), 108
 - ODVA (Open DeviceNet Vendor Association), 108
 - Open Sound System (OSS), 89
 - open source (开放源码)
 - ALSA project (ALSA 计划), 89
 - Apache HTTP servers (Apache HTTP 服务器), 309
 - BDM debugger (BDM 调试器), 343
 - BlueZ project (BDM 计划), 105
 - bootloaders listed (列出的引导加载程序), 255, 256
 - CAN projects (CAN 计划), 108
 - CompactPCI bus (CompactPCI 总线), 77
 - distribution considerations (发行套件的考虑), 38
 - embedded Linux (嵌入式 Linux), 27
 - hardware projects (硬件计划), 364

- home automation (家庭自动化), 86
- IDEs, 156
- licensing and (授权与), 27
- Modbus projects (Modbus 计划), 110
- movement for (的活动), 11
- Net-SNMP, 298
- OpenSSH, 303
- PPC support (对 PPC 的支持), 67
- support restrictions (支持受到限制), 64
- U-Boot, 260
- virtual machines (虚拟机), 150
- Open Source Initiative (OSI)(开放源码协会), 149
- OpenBT, 105
- Opencores project (Opencores 计划), 364
- OpenGroup, 30, 364
- OpenOBEX, 103, 105
- OpenOffice, 67, 71
- OpenSSH, 181, 246, 303-308
- OpenSSL, 303, 305
- /opt directory (/opt 目录), 179
- oscilloscopes (示波器), 338, 342
- OSI (Open Source Initiative)(开放源码协会), 149
- OSS (Open Sound System)(开放音效系统), 89
- P**
- PAGE_CACHE_SIZE, 238
- panic() function (panic()函数), 176
- panic_notifier_list, 176
- parallel ports (并口)
 - hardware support (所支持的硬件), 77, 82, 83
 - kernel support options (内核支持选项), 167
 - process control (过程控制), 85
- partitions (分区)
 - bootloader image (引导加载程序的映像), 219
 - CF devices (CF 设备), 229, 287
 - CFI flash and (CFI 闪存与), 215-217
 - creating filesystems in (在其中建立文件系统), 242
 - DOC, 223, 227, 228
 - erase blocks (抹除区块), 217
 - mounting (挂载), 242
 - MTD subsystem (MTD 子系统), 208
- patch utility (patch 公用程序), 249, 250
- patches (补丁)
 - GRUB interrupt handler (GRUB 中断处理例程), 271
 - kernel considerations (内核的考虑), 165
 - retrieving code by date (根据日期取回程序代码), 272
 - worksheet (工作单), 348
- patent issues (专利问题), 31-37, 96
- PATH environment variable (PATH 环境变量), 193, 198
- PC/104 bus (PC/104 总线), 73, 74
- PC/104 Consortium (PC/104 联盟), 74
- PC/104 single board computer (SBC)(单板计算机), 18
- PCI Industrial Computer Manufacturer's Group (PCIMG)(PCI 工业计算机制造商组织), 75
- PCI (Peripheral Component Interconnect) bus (PCI (外围部件互连) 总线), 72, 74, 75
- PCMCIA bus (PCMCIA 总线)
 - 802.11 cards (802.11 卡), 104
 - CF cards and (CF 卡与), 228
 - CompactFlash adapters (CompactFlash 配接卡), 97
 - FTL user module and (FTL 用户模块与), 96
 - hardware support (所支持的硬件), 73
 - LILO and (LILO 与), 267
- PDA (Personal Digital Assistant)(个人数字助理), 20, 70
- pdisk utility (pdisk 公用程序), 287
- PDQ print management package (PDQ 打印管理套件), 90
- performance analysis (性能分析)
 - code coverage (程序代码涵盖范围), 331-333
 - interrupt latency (中断等待时间), 337-339
 - kernel profiling (内核统计), 335-337
 - process profiling (进程统计), 330, 331
 - system monitoring (系统监控), 111, 112
 - system profiling (系统统计), 333-335
- Peripheral Component Interconnect (见 PCI)
- Perl programming language (Perl 程序语言), 86, 151-153, 335
- permissions (见 security)
- per-process statistics (每个进程的统计资料), 334
- persistent storage (永久储存), 233, 234
- Personal Digital Assistant (PDA)(个人数字助理), 20, 70

- physical address space (物理地址空间), 62, 63
 - PICMG (PCI Industrial Computer Manufacturer's Group)(PCI 工业计算机制造商组织), 75
 - piconets, 104
 - ping utility (ping 公用程序), 106, 294, 296
 - pivot_root() system call (pivot_root()系统调用), 245
 - PLCs (programmable logic controllers)(可编程逻辑控制器), 42, 85
 - PMON (Prom Monitor) bootloader (PMON 引导加载程序), 256, 257, 259
 - PNP (Plug and Play) devices (即插即用设备), 72
 - pointer devices (指位设备), 88
 - portmapper service (portmapper 服务), 266
 - PostScript (PS) format (PostScript 格式), 90
 - POTS (plain old telephone system)(旧式电话系统), 83
 - power failure CompactFlash devices (电源故障 CompactFlash 设备), 98
 - POWER (Performance Optimization With Enhanced RISC), 67
 - power-down reliability (断电可靠性)
 - ext2 and (ext2 与), 242
 - filesystems and (文件系统与), 233
 - JFFS2, 235
 - JFFS2 user module (JFFS2 用户模块), 95
 - PowerPC (PPC)
 - architecture overview (架构概述), 67, 68
 - bootloader comparison (引导加载程序的比较), 256
 - diet libc support (对 diet libc 的支持), 146
 - embedded system survey (嵌入式系统的调查), 22
 - embutils, 197
 - gdb debugger (gdb 调试器), 316
 - GNU toolchain (GNU 工具链), 117
 - hardware support options (硬件支持选项), 167
 - host build example (主机建立范例), 118
 - kernel (内核), 164, 167, 169, 174
 - PCMCIA support (对 PCMCIA 的支持), 73
 - U-Boot, 257
 - PowerPC Reference Platform (PReP)(PowerPC 参考平台), 68
 - PPCBoot (见 U-Boot)
 - PPR print management package (PPR 打印管理套件), 90
 - PREFIX environment variable (PREFIX 环境变量), 127, 143
 - PREFIX variable (PREFIX 变量) (make 命令), 120, 192, 299
 - PReP (PowerPC Reference Platform)(PowerPC 参考平台), 68
 - printenv command (printenv 命令), 277, 279
 - printf(), 48
 - printing (打印)
 - hardware support (所支持的硬件), 90, 91
 - parallel port I/O (并口输出/入), 83
 - privilege separation user (不涉及特权的用户), 308
 - /proc directory (/proc 目录), 180
 - /proc filesystem (/proc 文件系统), 63, 333
 - process automation (程序自动化), 84, 85
 - process profiling (进程统计), 330-335
 - processors (处理器)
 - architecture overview (架构概述), 64
 - bootloader variety and (各种引导加载程序与), 254
 - configuration options (配置选项), 125
 - constraints below 41bits (低于 32 位的限制), 16
 - hardware worksheet (硬件工作单), 347
 - kernel name selection and (内核名称选择与), 167
 - kernels appropriateness for (适合…的内核), 164
 - uClibc support (对 uClibc 的支持), 140
 - procps package (procps 套件), 333
 - profile= boot parameter (profile= 引导参数), 336
 - programmable logic controllers (PLCs)(可编程逻辑控制器), 42, 85
 - project identification worksheet (计划识别信息工作单), 346
 - project workspace (计划工作空间), 113-115
 - protocols (见 specific protocols)
 - ps utility (ps 公用程序), 333
 - ptrace() system call (ptrace()系统调用), 316, 322
 - publication resources (出版物资源), 363
 - public-key cryptography (公钥密码学), 303
 - Python programming language (Python 程序语言), 153-154
- Q**
- queuing printers and (队列打印机与), 90

R

RACSI (Remote ATV Control at ISS)(ISS 端 ATV 控制), 21

Radio Frequency Interference (RFI)(射频干扰), 107

RAM disks (RAM 硬盘)

- booting with (以 (RAM disk) 引导), 286-287
- copying image to flash (复制映像至闪存), 237
- filesystems and (文件系统与), 233, 234, 242-244

RAM (random access memory)(随机存取内存)

- CFI flash and (CFI 闪存与), 214
- flash location and (flash 的位置与), 275
- generic requirements (一般需求), 57
- hardware worksheet (硬件工作单), 347
- MTD support (MTD 支持), 209
- root filesystem and (根文件系统与), 58 (另见 memory)

ranlib utility (ranlib 公用程序) (binutils), 126, 135

readelf utility (readelf 公用程序) (binutils), 126, 184, 317

read-only block (只读区块), 97, 208

readprofile utility (readprofile 公用程序), 336

read/write access rights (读写存取权限), 157

real-time Linux (实时 Linux), 14

Real-Time Linux Foundation, 30, 364

Red Hat

- access rights (存取权限), 157
- crosgcc mailing list (crosgcc 邮递论坛), 123
- Cygwin environment (Cygwin 环境), 52
- MIPS support (对 MIPS 的支持), 69
- overview (概述), 28
- SourceNavigator IDE, 156
- survey findings (调查的结果), 22

RedBoot bootloader (RedBoot 引导加载程序)

- comparison (比较), 256
- features (特性), 260
- M68k, 257
- SuperH, 257

reiserfs journalling filesystem (reiserfs 日志型文件系统), 236, 242

remote administration (远程管理), 297-300, 309

Replace, 246

reset command (reset 命令), 291

resources (资源)

- books (书籍), 362, 363
- GNU toolchain (GNU 工具链), 122, 123
- online (在线), 361, 362
- open source projects (开放源码计划), 364
- organizations (组织), 363
- publications (出版物), 363

RFC1051 protocol (RFC1051 协议), 109

RFC1201 protocol (RFC1201 协议), 109

RFCOMM protocol (RFCOMM 协议), 106

RFCOMM (BlueZ), 106

RFI (Radio Frequency Interference)(射频干扰), 107

Roll-Your-Own survey (Roll-Your-Own 调查), 22

ROLO (ROmable LOader) bootloader (ROLO 引导加载程序), 256, 258

ROM (只读内存)

- booting from (引导自), 258
- CFI flash and (CFI 闪存与), 214
- hardware worksheet (硬件工作单), 347
- MTD support (对 MTD 的支持), 209
- ROMFS cautions (ROMFS 的注意事项), 234

ROMFS (ROM file system) cautions (ROMFS 的注意事项), 234

/root directory (/root 目录), 179

root filesystem (根文件系统)

- basic structure (基本结构), 178-181
- bootloaders (引导加载程序), 60, 61
- building (建立), 45, 47
- C library and (C 链接库与), 133
- CRAMFS, 237-240
- custom applications (定制应用程序), 198
- debugging (调试), 315
- development/production differences (开发/生产的差异), 48
- device files (设备文件), 188-189
- disk filesystem over NFTL (NFTL 上的磁盘文件系统), 241, 242
- disk filesystem over RAM disk (RAM disk 上的磁盘文件系统), 242-244
- generic requirements (一般需求), 57
- init program and (init 程序与), 198
- JFFS2, 240, 241
- kernel and (内核与), 58, 187
- libraries (链接库), 181-187
- live updates (在线更新), 245-252
- minit and (minit 与), 204

- NFS-mounted (经 NFS 挂载), 53、61、265-266
- selecting filesystem type for (选择文件系统类型), 232-237
- start_kernel() function (start_kernel()函数), 59
- SYSM module (SYSM 模块), 44
- system applications (系统应用程序), 189-198
- system initialization (系统初始化), 198-204
- System V init program (System V init 程序), 200
- TMPFS, 244、245
- top-level directories (顶层目录), 179
- worksheet (工作单), 349
- writing image to flash (将映像写入闪存), 237
- root hub (USB)(根集线器), 78、79
- root privileges (root 的权限), 181、194
- RPC
 - inetd and uClibc (inetd 与 uClibc), 295
 - xinetd and (xinetd 与), 296
- RPM (RPM Package Manager), 249
- RS232 interface (RS232 接口)
 - I/O support (对输出/入的支持), 82
 - linked setup (链接式规划), 53
 - Modbus protocol and (Modbus 协议与), 110
 - terminal emulation and (终端机仿真与), 157
- RSA keys (RSA 密钥), 307
- rsh shell, 246、247
- rsync utility (rsync 公用程序), 246-248
- RTAI project (RTAI 计划)
 - Adeos nanokernel and (Adeos nanokernel 与), 37
 - ARM support (对 ARM 的支持), 66
 - GPL licensing and (GPL 的授权与), 35
 - MIPS support (对 MIPS 的支持), 69
 - PowerPC support (对 PowerPC 的支持), 67
 - real-time Linux (实时 Linux), 14
 - software watchdog (软件 watchdog 定时器), 112
- RTLinux
 - ARM support (对 ARM 的支持), 66
 - licensing (授权), 36
 - PowerPC support (对 PowerPC 的支持), 67
 - project (计划), 14
 - SuperH support (对 SuperH 的支持), 70
- RTNet, 44
- RTU (Modbus messaging format)(Modbus 通讯格式), 110
- run command (run 命令), 279
- S**
 - SAE (Society of Automotive Engineers), 108
 - saveenv command (saveenv 命令), 279
 - SBC (single board computer)(单板计算机), 18、78
 - /sbin directory (/sbin 目录), 180
 - SCADA (System Control and Data Acquisition) protocol converter (SCADA 协议转换器), 20
 - scripts (脚本)
 - ad hoc updating scripts (特制更新脚本), 249-252
 - CGI scripting (撰写 CGI 脚本), 309
 - creating boot (建立引导), 279、280
 - trace helpers (追踪辅助程序), 327
 - SCSI (Small Computer Systems Interface)(小型计算机系统接口)
 - CF cards and (CF 卡与), 228
 - fdisk utility (fdisk 公用程序), 98
 - hardware support (所支持的硬件), 77、78
 - IEEE1394 differences (IEEE1394 的差异), 80
 - LILO and (LILO 与), 267、268
 - U-Boot and (U-Boot 与), 260
 - SCSL (Sun Community Source License)(Sun 社群源码授权书), 149
 - SDP (Service Discovery Protocol)(服务找寻协议), 106
 - SDPd (BlueZ), 106
 - SDS (Smart Distributed System) protocol (SDS 协议), 108、109
 - Secondary Program Loader (SPL), 223、270
 - security (安全)
 - changing permissions (变更使用权), 181
 - gdb cautions (gdb 的注意事项), 320
 - root login cautions (root 登入的注意事项), 115
 - secure authorization (安全授权), 296
 - secure communication (安全通讯), 303-309
 - worksheet (工作单), 350
 - Sega game console (Sega 游戏机), 70
 - sensord daemon (sensord 服务), 112
 - serial infrared (SIR)(串行红外线), 101
 - serial ports (串口)
 - data loss (资料漏失), 223

- debugging host/target systems (进行主机/目标系统的调试), 55
- embedded system example (嵌入式系统范例), 18
- gdb, 319, 320
- hardware support (对硬件的支持), 81, 82
- modems as (将调制解调器视为), 83
- process control (过程控制), 85
- serial dongles (串行转换器), 102
- terminal emulators (终端机仿真程序), 157-159
- set remotebaud command (set remotebaud 命令)(gdb), 321
- set solib-absolute-prefix command (set solib-absolute-prefix 命令)(gdb), 321
- setenv command (setenv 命令), 278
- SH project (SH 计划), 167, 259
- shadow password support (对影子密码的支持), 195, 196, 304, 308
- SHARED_LIB_LOADER_PATH variable (SHARED_LIB_LOADER_PATH 变量)(uClibc), 143
- Sharp Zaurus, 20
- sh-boot bootloader (sh-boot 引导加载程序), 256, 257, 259
- shells
 - BusyBox, 193, 202
 - performing updates (进行更新), 251, 252
 - rsh shell, 246, 247
 - ssh shell, 246, 247
- .shstrtab section (.shstrtab 段)(ELF 二进制), 319
- Simple Network Management Protocol (简易网络管理协议)(SNMP), 297-300
- Simputer project (Simputer 计划), 364
- Single Unix Specification (SUS), 30
- SIR (serial infrared)(串行红外线), 101
- size utility (size 公用程序)(binutils), 126
- Small Computer Systems Interface (见 SCSI)
- SMBus (System Management Bus), 81
- SNMP (Simple Network Management Protocol), 297-300
- snmpd utility (snmpd 公用程序), 300
- snmpget utility (snmpget 公用程序), 299
- SoC (System-on-Chip)(单晶系统), 65, 78
- Society of Automotive Engineers (SAE), 108
- soft real-time system (软实时系统), 17
- software (软件)
 - availability for i386 (i386 的可用性), 65
 - distribution considerations (发行套件的考虑), 34
 - Linux and (Linux 与), 23, 24, 25
 - package management tools (套件管理工具), 249
 - running vs. modifying (执行与修改), 33
- solid state storage media (固态储存媒体), 60, 61
- sound hardware support (声音所支持的硬件), 89
- SourceForge project (SourceForge 计划), 26, 73, 105, 259
- SourceNavigator (IDE), 156
- SPL (Secondary Program Loader), 223, 270
- spooling system print process and (spooling 系统打印过程与), 90
- spread spectrum frequency hopping (扩频跳频), 104
- S-Record format (Motorola)(S-Record 格式), 281, 285
- SSH protocol (SSH 协议), 43, 303-309, 320
- ssh shell, 246, 247
- SSL (Secure Socket Layer) protocol (SSL 协议), 303
- .stab section (.stab 段)(ELF 二进制), 317, 319
- stab (symbol table)(符号表), 317
- .stabstr section (.stabstr 段)(ELF 二进制) 309, 311
- _start symbol (_start 符号), 139
- start_kernel() function (start_kernel()函数), 59
- state machines control modules and (状态机控制模块与), 42
- static linking (静态链接)
 - Boa, 309, 310
 - BusyBox, 191
 - copyright laws (版权法), 33
 - diet libc, 148
 - DOSTATIC flag (DOSTATIC 标志), 192, 195
 - embutils, 197
 - glibc setup option (glibc 规划选项), 131, 132
 - LDFLAGS option (LD_FLAGS 选项), 326
 - libraries and (链接库与), 59
 - microp Perl, 152
 - rsync, 246
 - strace and uClibc (strace 与 uClibc), 323
 - thttpd, 311

- udhcp, 313
- xinetd, 296
- statistics (统计资料), 333, 334
- stepper motors (步进马达), 18
- STMicroelectronics (SGS-Thomson Microelectronics), 70
- storage devices (存储设备)
 - boot configuration setups (引导配置的规划), 61
 - build-tools directory cleanup, 137
 - disk devices (磁盘设备), 228-231
 - DiskOnChip, 219-228
 - embedded systems and (嵌入式系统与), 205
 - generic requirements (一般需求), 57
 - hardware support (所支持的硬件), 91-99
 - hardware worksheet (硬件工作单), 347
 - initrd images (initrd 映像), 242
 - linked setup and (链接式规范与), 53
 - Linux workstations (Linux 工作站), 51
 - log files and (日志文件与), 310
 - MTD subsystem (MTD 子系统), 206-214
 - Native CFI Flash (原生 CFI 闪存), 214-219
 - persistent storage (永久储存), 233, 234
 - removable storage setup (移动存储设备的规划), 53
 - root filesystem (根文件系统), 58
 - setting up (设置), 47
 - structure for access (存取的结构), 58
 - swapping (交换), 231
 - worksheet (工作单), 349
 - writing CRAMFS image (写入 CRAMFS 映像), 240
 - (另见 solid state storage media)
- strace tool (strace 工具程序), 322
- stringent time constraints (严格时限), 16
- strings utility (strings 公用程序)(binutils), 126
- strip utility (strip 公用程序)
 - binutils package (binutils 套件), 126
 - ELF binary and (ELF 二进制文件与), 319
 - libraries (链接库), 185
 - reducing binary sizes (降低二进制的大小), 317
 - relocating (重新配置), 135
 - telnetd cautions (telnetd 的注意事项), 301
- .strtab section (.strtab 段)(ELF 二进制), 319
- Sun Community Source License (SCSL), 149
- SuperH processor (SuperH 处理器)
 - appropriate kernel location (最适合的内核供应网站), 164
 - architecture overview (架构概述), 70
- bootloader comparison (引导加载程序的比较), 256
- GNU toolchain version combinations (GNU 工具链的版本组合), 117
- sh-boot and RedBoot (sh-boot 与 RedBoot), 257
- watchdog timers for (watchdog 定时器), 112
- swapping storage devices and (交换 储存设备与), 231
- symbolic debugging (符号调试。见 debugging gdb tool)
- symbolic links (符号链接)
 - BusyBox, 193, 201
 - creating to relocated binaries (建立可重定位的二进制文件), 136
 - /dev directory (/dev 目录), 189
- embutils, 198
- glibc package (glibc 套件), 182, 183
- kernel configuration (内核的配置), 169
- OpenSSH, 305
- rsync updating utility (rsync 更新公用程序), 246
- TinyLogin, 194, 195
- .symtab section (.symtab 段)(ELF 二进制), 319
- YSM (system management) module (系统管理模块)
 - Boa, 309-311
 - compacting (缩小尺寸), 44
 - DAQ module and (DAQ 模块与), 41
 - dynamic configurations (动态配置), 313
 - embedded system (嵌入式系统), 40, 43-44
 - HTTP and (HTTP 与), 309
 - keepalive signals (keepalive 信号), 44
 - netkit-base example (netkit-base 范例), 294-296
 - netkit-telnet, 301, 302
 - Net-SNMP package (Net-SNMP 套件), 298-300
 - OpenSSH, 304-308
 - real-time kernels (实时内核), 44
 - httpd, 311, 312
 - udhcp, 313, 314
 - utelnetsd, 302
 - xinetd build (建立 xinetd), 296, 297
- system applications (系统应用程序), 189-198, 349

system initialization (系统初始设定)
 panic function registration (panic 函数的注册), 176
 RAM disks and (RAM 磁盘与), 242
 root filesystem and (根文件系统与), 198-204
 worksheet (工作单), 349
 system libraries installing on root filesystem (系统链接库安装在根文件系统上), 181-187
 System Management Bus (SMBus), 81
 system startup (系统的启动), 59, 60, 223
 System V init program (System V init 程序), 198, 199, 201
 SYSTEM_DEVEL_PREFIX variable (SYSTEM_DEVEL_PREFIX 变量) (uClibc), 143
 System.map file (System.map 文件), 173
 System-on-Chip (SoC)(单晶系统), 65, 78
 systems (系统)
 component determination (组件的决定), 45, 47
 configuration options (配置选项), 125
 defined (定义), 13
 monitoring (监控), 111, 112
 multicomponent (多组件), 39-44
 rebooting (重引导), 176, 202
 statistics (统计资料), 334

T

tar command (tar 命令), 124
 tar-bzip2 file (经 tar-bzip2 压缩的文件), 124
 target command (target 命令)(gdb), 321
 TARGET environment variable (TARGET 环境变量), 120, 127
 target remote command (target 远程命令), 320
 target systems (目标板系统)
 creating (建立), 45-47
 debugging (调试), 55, 56, 316
 defined (定义), 14
 developing (开发), 52-55
 gdb constraints (gdb 的限制), 317
 GNU toolchain (GNU 工具链), 116, 117
 installing MTD utilities (安装 MTD 公用程序), 213, 214
 network login (以网络登录), 300
 RAM disks and (RAM 磁盘与), 234
 self-hosting (自主), 234, 237
 TARGET variable (TARGET 变量), 120

testing connections (测试连接), 48
 (另见 host systems)
 TARGET_ARCH variable (TARGET_ARCH 变量) (make 命令), 192
 TARGET_PREFIX variable (TARGET_PREFIX 变量), 120, 121
 TCP/IP
 embedded system example (嵌入式系统范例), 40
 gdb servers (gdb 服务器), 319, 320
 host/target debugging setups (主机/目标板调试规划), 55
 Modbus protocol (Modbus 协议), 110
 remote management with SNMP (通过 SNMP 进行远程管理), 297
 time constraints and (时限与), 44
 Telephony Control protocol Specification Binary, 105
 Telnet protocol (Telnet 协议), 297, 300-303
 telnetd daemon (telnetd 服务), 301, 302
 terminal emulators (终端机仿真程序)
 background (背景), 157-162
 C-Kermit, 160-162
 mimcom, 157, 159
 sending image file to target (将映像文件送至目标板), 284
 U-Boot and (U-Boot 与), 275, 291
 UUCP cu, 157, 159, 160, 285
 worksheet (工作单), 347
 Terminate and Stay Resident (TSR) program (驻留程序), 270
 testing (测试)
 disabling netkit-base (停用 netkit-base), 295
 host/target connection (主机/目标板的连接), 48
 .text section (.text 段) (ELF 二进制), 317
 TFTP (Trivial File Transfer Protocol)(简易文件传输协议)
 booting with (使用 (TFTP) 引导), 282-283
 linked setup (链接式规划), 53
 network boot (网络引导), 61, 257, 261
 setting up daemon (设置服务), 263-265
 U-Boot image into RAM (将 U-Boot 映像载入 RAM), 290
 tftpboot command (tftpboot 命令), 284
 thttpd, 309, 311, 312
 time constraints (时限), 16, 17, 44
 timestamps (时时戳), 238
 TinyLogin, 194-196, 251

TinyTP (Tiny Transport Protocol), 102
TiVo system (TiVo 系统), 67
Tkinter interface (Tkinter 接口), 154
/tmp directory (/tmp 目录), 179, 181, 245
TMPFS mounting directories (TMPFS 挂载目录), 236, 244, 245
top utility (top 公用程序), 333
TQM860L board (TQM860L 实验版), 216, 217
trace command (trace 命令), 328
trace daemon (trace 服务), 326-327
traceanalyze command (traceanalyze 命令), 330
tracedump command (tracedump 命令), 330
traceview command (traceview 命令), 328
tracevisualizer command (tracevisualizer 命令), 328, 329, 330
tracing (追踪), 322-330
transducers (传感器), 40, 84
trap daemon (trap 服务)(SNMP), 299
Trivial File Transfer Protocol (见 TFTP)
TrueFFS tools (TrueFFS 工具程序)(M-Systems), 219
TuxScreen project (TuxScreen 计划), 364
TV Linux Alliance, 30, 364

U

UARTs (Universal Asynchronous Receiver-Transmitters)(通用异步收发器), 82, 83
U-Boot bootloader (U-Boot 引导加载程序)
ARM and (ARM 与), 257
binary images (二进制映像), 280-281, 283-285
booting (引导), 275-277, 282-283
booting from CF devices (自 CF 设备引导), 287-290
booting with RAM disk (使用 RAM 磁盘引导), 286-287
CF device partitions (CF 设备分区), 229
command help (命令辅助说明), 277
comparison (比较), 256
compiling and installing (编译及安装), 274-275
emulation constraints (仿真的限制), 157
environment variables (环境变量), 277
features (特性), 260
MIPS and (MIPS 与), 257
PowerPC and (PowerPC 与), 257
update cautions (更新的注意事项), 290
updating (更新), 290-292
U-Boot's environment variables (U-Boot 的环境变量), 176, 279
uClibc library (uClibc 链接库)
Boa, 309
BusyBox and (BusyBox 与), 192
features (特性), 140-146
file dependencies (文件的依赖关系), 250
FreeSSH constraints (FreeSSH 的限制), 309
inetd support (对 inetd 的支持), 295
ldd command and (ldd 命令与), 184
microperl and (microperl 与), 152
Net-SNMP and (Net-SNMP 与), 298, 299
OpenSSH and (OpenSSH 与), 304
patch utility (patch 公用程序), 250
Python considerations (Python 的考虑), 154
rsync utility (rsync 公用程序), 246
shell script for updates (用于更新的 shell 脚本), 251
strace static link (strace 静态链接), 323
telnetd, 301
tftp, 311
TinyLogin, 194, 195
udhcp, 313
utelnetd, 302
xinetd constraints (对 xinetd 的限制), 296
uClinux project (uClinux 计划), 64, 364
udhcp project (udhcp 计划)(BusyBox), 313
UI (user interface) module (用户接口模块)
DAQ module and (DAQ 模块与), 41
dynamic configurations to (对 (UI 模块的) 动态配置设定), 313
embedded system example (嵌入式系统范例), 40, 44
JFFS2 and (JFFS2 与), 236
patch utility (patch 公用程序), 251
SYSM module and (SYSM 模块与), 43
system memory layout (系统内存的配置), 62
as X terminals (如 X 终端机), 44
UID field (UID 字段), 238
Universal Asynchronous Receiver-Transmitters (UARTs)(通用异步收发器), 82, 83
Unix workstations (Unix 工作站), 51, 52
unlock utility (MTD), 210
updating (更新)
live updates (在线更新), 245-252
U-Boot, 290-292

USB (Universal Serial Bus) interface (通用串行总线接口)

- CF cards and (CF 卡与), 228
- hardware support (所支持的硬件), 78, 79
- IEEE1394 differences (与 IEEE1394 的差异), 80
- kernel support options (内核支持选项), 167
- LILO and (LILO 与), 267
- Linux I/O device support (对 Linux 输出入设备的支持), 81
- USB dongles (USB 转换器), 102

USB-IF (USB Implementers Forum)(USB 开发论坛), 78

user accounts (用户账号), 310, 312

USE_SYSTEM_PWD_GRP variable
(USE_SYSTEM_PWD_GRP 变量), 195

USE_SYSTEM_SHADOW variable
(USE_SYSTEM_SHADOW 变量), 195

/usr directory (/usr 目录), 180

utelnetsd package (utelnetsd 套件), 302

util-linux package (util-linux 套件), 336

UUCP (Unix to Unix CoPy) cu, 157, 159, 160, 285

V

value-added packages (增值套件), 38

/var directory (/var 目录), 179, 180, 181

VDC (Venture Development Corporation), 22

vendor support (厂商的支持)

- ARM, 66, 67
- CAN, 107
- CompactPCI bus (CompactPCI 总线), 77
- DAQ packages (DAQ 套件), 84
- distributions and (发行套件与), 38
- I²C bus (I²C 总线), 80
- independence (不依赖), 26
- IrDA, 100
- MIPS, 69
- Motorola 68000 processors (Motorola 68000 处理器), 71
- PowerPC architecture (PowerPC 架构), 67
- process control (过程控制), 85
- VME bus (VME 总线), 75

Venture Development Corporation (VDC), 22

versions (版本)

- EXTRAVERSION variable
(EXTRAVERSION 变量), 170

firmware (固件), 220, 221

kernels (内核), 163, 164, 165

LILO, 267

naming conventions (命名习惯), 170

NTFL formatting (进行 NTFL 的格式化), 226

tracking (追踪), 304

worksheet (工作单), 348

VfIR (very fast infrared)(极快速红外线), 101

vi (IDE), 156

ViewML, 29

virtual addresses (虚拟地址), 62, 63

virtual machines (虚拟机), 150

visualization tool (可视化工具), 325, 327, 328

VME bus (VME 总线), 71, 74, 75

vmlinux file (vmlinux 文件), 173

VxWorks (WindRiver), 21, 322

W

watchdog timers (watchdog 定时器), 111, 112

wear leveling (损耗平衡), 95, 235

web content (网页的内容), 309-312

Windows workstations (Windows 工作站), 52

WindowsCE, 69

WinModem, 83, 84

wireless technologies (见 see Bluetooth; IEEE 802.11; IrDA)

worksheet embedded Linux systems (工作单嵌入式 Linux 系统), 345-352

workspace (工作空间), 113-115, 120-121

workstations (工作站), 50, 51, 52

X

X terminals (X 终端机), 44, 258

X Window System (X 窗口系统)

- graphical interface (图形接口), 89
- JDK and JRE (JDK 与 JRE), 149
- kernel configuration (内核配置), 169
- xconfig script (xconfig 脚本), 124

X10 corporation (X10 公司), 86

X10 Power Line Carrier (PLC) protocol (X10 电源线载子协议), 86, 87

x86 processor (x86 处理器)

- architecture overview (架构概述), 65, 66
- bootloader comparison (引导加载程序的比较), 256
- bzImage target (bzImage 建立目标), 172
- diet libc support (对 diet libc 的支持), 146

- DiskOnChip devices (DiskOnChip 设备), 219
 - embedded system survey (嵌入式系统的调查), 22
 - GRUB and (GRUB 与), 272
 - ISA support (对 ISA 的支持), 72
 - kernel (内核), 164, 167
 - system startup process (系统启动程序), 223
 - xconfig command (xconfig 命令), 124
 - XEmacs (IDE), 156
 - XFS journalling filesystem (XFS 日志型文件系统), 242
 - xinetd super-server (xinetd 超级服务器)
 - features (特性), 296, 297
 - Red Hat-based (以 Red Hat 为基础), 264
 - telnetd and (telnetd 与), 302
 - TFTP service (TFTP 服务), 264
 - XIP (eXecute In Place)(本地执行), 95
- Y**
- Yellow Dog Linux, 67
- Z**
- zImage file (zImage 文件), 173
 - zlib compression library (zlib 压缩链接库), 213, 303, 304, 308

[General Information]

OS=Linux

OS=

OS=407

SS=0

OS=

